AFRL-RI-RS-TR-2015-176

CHARACTERIZING AND IMPLEMENTING EFFICIENT PRIMITIVES FOR PRIVACY-PRESERVING COMPUTATION

GEORGIA INSTITUTE OF TECHNOLOGY

JULY 2015

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2015-176 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /
CARL R. THOMAS
Work Unit Manager

/ S /
MARK H. LINDERMAN
Technical Advisor, Computing
& Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
JULY 2015	FINAL TECHNICAL REPORT	MAY 2011 – MAR 2015

4. TITLE AND SUBTITLE

CHARACTERIZING AND IMPLEMENTING EFFICIENT PRIMITIVES FOR PRIVACY-PRESERVING COMPUTATION

5a. CONTRACT NUMBER
FA8750-11-2-0211

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)

Patrick Traynor, Kevin Butler

5d. PROJECT NUMBER
PROC

5e. TASK NUMBER
ED

5f. WORK UNIT NUMBER
GA

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Georgia Institute of Technology
North Ave NW
Atlanta, GA 30332

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER
AFRL-RI-RS-TR-2015-176

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

While garbled circuits have been known for nearly 30 years, efficient realizations of such schemes have only become possible recently. However, their use on mobile devices, where the nature of applications are different and the use of context sensitive information is the norm and not the exception, has just begun to be assessed. The goal of this project is simple – allow mobile devices to take part in secure computation without significant degradation in performance and security when compared to their desktop counterparts. When taken as a whole, our work has moved the reality of SFE on mobile devices from barely possible to equivalent in performance and security when compared against modern two-party schemes. This document discusses the details of our advances, tangible improvements and remaining challenges.

15. SUBJECT TERMS
Garbled Circuit, Cell Phone, Encryption, Cryptography, Mobile Devices, Secure Multi-party Computation (SMC), Secure Function Evaluation (SFE)

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			**CARL R. THOMAS**
U	U	U	UU	143	19b. TELEPHONE NUMBER (*Include area code*) **N/A**

Table of Contents

Table of Figures

Summary

Secure Function Evaluation (SFE) holds the promise of protecting data while still allowing important computation to be executed upon it. However, the primitives making such computation possible are extremely expensive, and have long been viewed as entirely outside of reach for all but the most powerful of computing platforms. This work focuses on enabling the use of SFE on mobile phones, the most widely deployed computing infrastructure in the world. This work demonstrates our progression from the complete inability to run even the most basic such computations to seamlessly participating in the execution of the largest created garbled circuits to date without any loss of security.

Introduction

The confluence of high-speed connectivity and device capability has led to the recent surge in mobile application development. While software common to desktop computing (e.g., word processing, email) exists in this space, the most popular mobile applications often provide services based on a user's current context (e.g., location, social interconnections, etc.). Such applications allow users to make more informed decisions based on their surroundings. However, these applications also regularly expose sensitive data to potentially untrusted parties.

Cryptographers have long worked to develop mechanisms that allow two parties to compute shared results without exposing either individual's sensitive inputs or requiring assistance from a trusted third-party. Such techniques are referred to as *Secure Function Evaluation* (SFE), and provide a set of powerful primitives for privacy- preserving computation. While garbled circuits have been known for nearly 30 years [3], efficient realizations of such schemes have only become possible recently. However, their use on mobile devices, where the nature of applications are different and the use of context sensitive information is the norm and not the exception, has just begun to be assessed.

The goal of this project is simple – allow mobile devices to take part in secure computation without significant degradation in performance and security when compared to their desktop counterparts. The reasons for this goal are numerous. First, mobile phones are used by more than six billion people across the globe every day. When compared to the two billion individuals who currently have access to traditional computing resources, these platforms by far represent the dominant form of computing available throughout the world. Second, mobile phones are increasingly being relied upon to store our most sensitive information, from a history of our locations and the people with whom we interacted, to personal conversations and financial information. This data is regularly exfiltrated and mined by untrustworthy third parties, creating uncontrollable digital footprints in our daily lives. Finally, mobile phones are increasingly being relied upon by members of industry and government (especially the military) as a critical platform for communication while outside of the office or within a theatre of war. Accordingly, efficient techniques for verifiably protecting the data that is generated, received, and transmitted by these devices are of great necessity to private citizens, companies and the government.

The goal of this project is difficult for many reasons. Chief among these is the comparative lack of processing ability available on mobile phones. With comparatively slow processors, limited memory, slow and often policy capped bandwidth and finite battery power, making SFE work at all would prove to be difficult.

This work represents nearly four years of concerted effort to make this goal a reality. As we began this work, it quickly became clear that none of the tools or theoretical constructions available to the traditional computing community (i.e., server-based computation, Yao's garbled circuits) would be sufficient to make SFE possible on mobile platforms. In fact, our early work shows that all but the most trivial problems were simply beyond the abilities of cutting edge techniques.

With this as our starting point, we first developed a series of efficient custom protocols that achieved the same ends as protocols written with garbled circuits. We also dedicated significant effort in our first year to developing a more efficient compiler for garbled circuits. While both of these efforts dramatically reduced the performance and bandwidth overheads of the best available garbled circuit techniques, neither was sufficient to meet our goal. Our solutions would need to dramatically improve performance and security guarantees in order to erase the gap between mobile and traditional SFE capabilities.

Our efforts in the second year focused on techniques designed to enable dramatic improvements in performance of SFE on mobile devices. In particular, we attempted to offload much of the work done on the mobile device in a secure computation to another, more powerful node. While the naïve approach would simply trust this third party to perform the operations on the mobile phone's behalf, our approach is able to offload the mobile's execution without any degradation in the traditional two-party SFE model while dramatically reducing total execution time.

Our efforts in our third year focused on improving performance across iterations of SFE protocols, allowing for the cost of certain operations to be amortized. We also expanded our outsourcing techniques to reduce execution time for some applications by a further 98%, while allowing us to execute the billion-gate circuits run between server class machines at the same security level. However further improvements were still necessary.

In our final year, we focused on techniques to substantially improve performance. On the outsourcing side, we developed a black box lifting technique that allows us to incorporate improvements made by other researchers directly into an outsourcing scheme, without having to prove their composition secure. Moreover, we built a principled compiler that, in addition to proving demonstrably more correct results than related work, does so orders of magnitude faster.

When taken as a whole, our work has moved the reality of SFE on mobile devices from barely possible to equivalent in performance and security when compared against modern two-party schemes. This document discusses the details of our advances, tangible improvements and remaining challenges.

Methods, Assumptions and Procedures

Efficient Mobile Oblivious Computation (EMOC)

Mobile applications increasingly require users to surrender private information, such as GPS location or social networking data. To facilitate user privacy when using these applications, Secure Function Evaluation (SFE) could be used to obliviously compute functions over encrypted inputs. The dominant construction for desktop applications is the Yao garbled circuit, but this technique requires significant processing power and network overhead, making it extremely expensive on resource-constrained mobile devices.

In this effort, we developed *Efficient Mobile Oblivious Computation* (EMOC), a set of SFE protocols customized for the mobile platform. Using partially homomorphic cryptosystems, we developed protocols to meet the needs of two popular application types: location-based and social networking. Using these applications as comparison benchmarks, we demonstrated execution time improvements of 99% and network overhead improvements of 96% over the most optimized garbled circuit techniques. These results showed that our protocols provide mobile application developers with a more practical and equally secure alternative to garbled circuits.

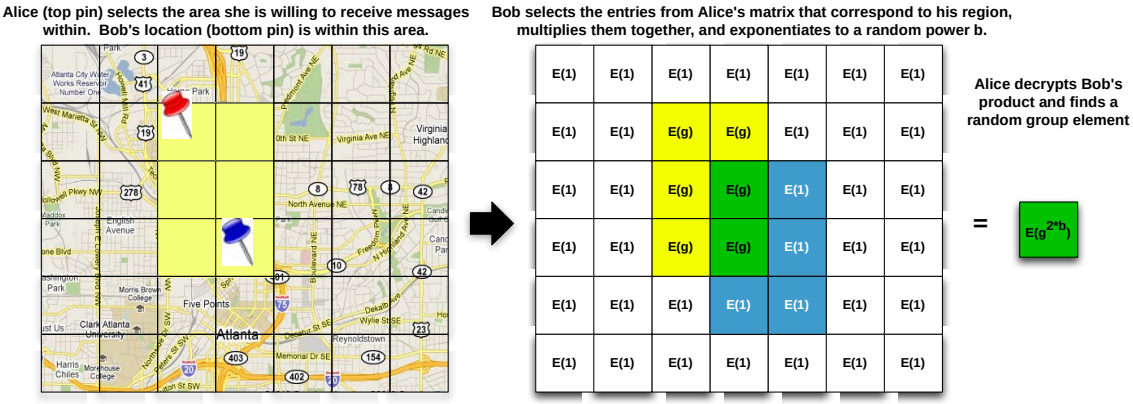


Figure 1: Location proximity testing. Areas of overlapping interest are determined by finding values other than "1" in the final cleartext.

Location-based messaging, especially for advertisements, has recently received significant attention. Beyond advertising based on location, it offers the potential for useful applications such as a proximity test to alert two people if they are close enough to arrange a meeting. It could also be combined with applications like Twitter to allow for location-based tweet filtering and following. However, these applications must query the physical location of a user, which could compromise the user's privacy. To resolve this information leakage, we present a protocol for securely computing when two users are within a chosen proximity of one another. While used in a specific application here, the protocol can be used in any location-based mobile application. The ability to specify an input region of any shape or size

allows the proximity test to provide a result at any desired granularity, from the same building to the same city.

Figure 1 shows our proposed solution. A user Alice creates an M x N matrix overlaid on an area of interest (e.g., her current city). Alice builds a location matrix with encryptions of '1' in every entry except those that correspond to the area she is willing to receive tweets within. In her travel area, she enters encryptions of generator 'g'. Bob selects the entries that correspond to his travel area, multiplies them together, exponentiates by a random blind, and returns the product to Alice. When Alice decrypts, she knows that: if the value is not '1', Bob's message is relevant to her. Otherwise, Bob's message is irrelevant to her location. As we prove in the paper, this protocol is secure in the semi-honest model, similar to the majority of Yao-based systems available at the time of the work.

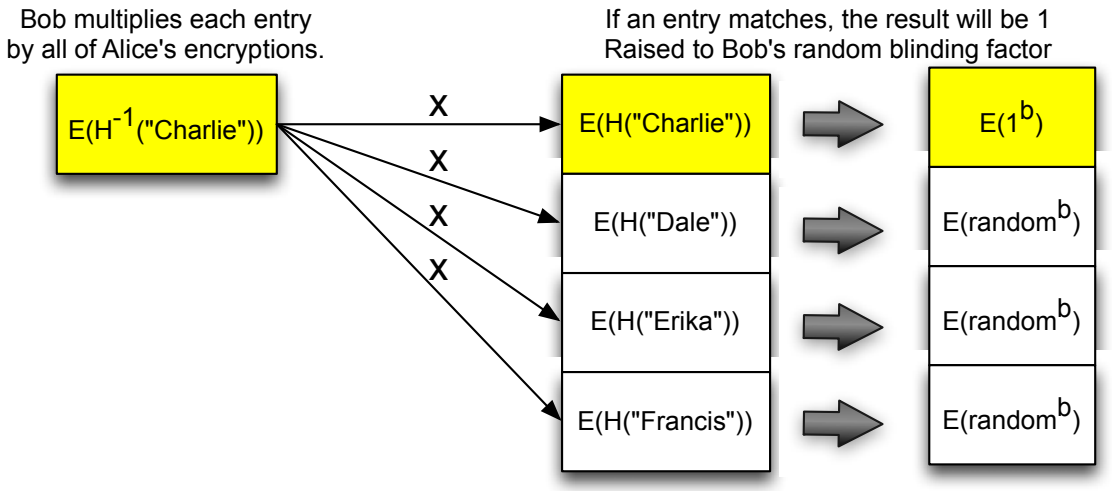


Figure 2: Private Set Intersection. Alice and Bob determine if their social sets intersect by comparing ciphertexts with additive homomorphic properties.

Social networking applications are a popular channel for communicating with a mobile device. However, they also create a potential channel to leak private information about a user's social life. If two mobile users were to meet at a party or conference, one might only want to allow the other into her social network based on the friends they already have in common. However, there is currently no mobile application that allows this without revealing both users' entire social graphs. This application offers a means for securely revealing only the friends common to both users while maintaining the privacy of the rest of both social graphs. Again, we couch our protocol in an application that is highly relevant to mobile users. However, the protocol can be used in general to compute the intersection of any two sets without revealing any element outside of the intersection.

Figure 2 provides our solution to this problem. Bob homomorphically multiplies each entry in his array by every entry in Alice's array. He then exponentiates by a unique blinding factor for all of the resulting values. Alice receives these values and

decrypts them. If an entry is equal to 1, Alice knows there is a match. While this approach has $O(n^2)$ theoretical complexity, we argue (and demonstrate) that its execution profile outperforms any garbled circuit implementation because of the expected size of real datasets.

Table 1: Performance profile of EMOC Applications. In both execution time and bandwidth overhead, our approach outperforms garbled circuit protocols.

Protocol	Input size	SFE scheme	Avg. exec. time (sec.)	Network use (KB)
Proximity Test	500 cells	EMOC	0.0165 (± 0.0001)	128.256
		OBDD	23.1480 (± 0.0351)	1,765.764
		Parallelized	26.2353 (± 0.0836)	1,854.049
		PAL	35.1888 (± 0.0487)	2,029.439
		Pipelined	11.1293 (± 0.0332)	603.497
		Fairplay	NA	NA
Private Set Intersection	20 friends	EMOC	3.7466 (± 0.0042)	107.520
		OBDD	124.4921 (± 0.2809)	2,879.016
		Parallelized	107.8990 (± 0.4249)	2,669.284
		PAL	130.7570 (± 0.2013)	3,025.966
		Fairplay	NA	NA
	16 friends	Pipelined	45.7061 (± 0.1254)	3,401.133

Table 1 shows the execution times and network overhead of the two proposed protocols. Our custom protocols far outperform all of the garbled circuits-based approaches, with improvements as high as 99% for execution time and 96% for bandwidth. Accordingly, our approaches are more appropriate for resource constrained mobile devices than the direct application of garbled circuits.

In spite of the significant performance improvements we gained through the use of custom protocols, a number of challenges remained. For instance, while we were able to outperform two specific protocols, our custom protocol approach does not "scale" easily and requires that new custom protocols are created for each potential application we want to implement. Such efficient protocols may not be available for all possible applications. Second, while our approach is robust in the semi-honest model, researchers were beginning to explore defenses against malicious adversaries. Finally, to demonstrate our progress over the entire PROCEED program, we felt the need to try and match the benchmarks used by other teams (e.g., AES). Accordingly, we determined that we would need to make substantial and fundamental advances in garbled circuits in order to support their use on mobile platforms.

Memory Efficient Garbled Circuit Generation for Mobile Devices

Given our desire to make the benchmark applications used by other PROCEED performers possible on mobile platforms, our next research effort attempted to make the use of garbled circuits more efficient. We note that this effort took place in parallel with our EMOC work, given the early realization that custom protocols would not be possible for all possible applications.

This new effort focused on developing a memory-efficient technique for generating the garbled circuits needed to perform secure function evaluation on smartphones. While numerous research initiatives have considered how to evaluate these circuits more efficiently, little work had focused on efficient generation. Such a consideration is particularly important given the significant memory constraints of mobile phones. We achieved this goal by creating the Pseudo Assembly Language (PAL), a mid-level intermediate representation (IR) compiled from Fairplay's SFDL high-level language, where each instruction represents a pre-built circuit. These templates allowed us to represent many complex instructions with a very limited amount of memory.

	Memory (KB)	
Program	Fairplay	FPPALC
Millionaires	658	296
Billionaires	1188	441
CoinFlip	1488	384
KeyedDB 16	NA	688
SetInter 2	10667	469
SetInter 4	NA	522
SetInter 8	NA	617
Levenshtein Dist 2	NA	392
Levenshtein Dist 4	NA	405
Levenshtein Dist 8	NA	429

Figure 3: Memory comparison of Fairplay and PAL (FPPALC). Note that many applications not possible using Fairplay are now possible on a mobile device.

Figure 3 shows a comparison of the memory profiles of Fairplay and our PAL system. The first important improvement over standard Fairplay is the significant reduction in memory required to execute applications such as the Millionaire's and Billionaire's problems (with savings of 55% and 63%, respectively). Second, and potentially more critically, the use of PAL enabled circuits that were previously too big to execute on a mobile device (Set Intersection for inputs of larger than size 2, all Edit Distance problems) to finally run on these systems.

While this work was a significant step forward for the execution of garbled circuits on mobile devices, many important innovations would need to continue to be made to ensure that such systems could actually perform relevant privacy preserving computation.

Secure Outsourced Garbled Circuit Execution for Mobile Devices

Our work up to this point made a number of points clear. In particular, the processing, bandwidth and memory constraints we encountered represented significant hurdles to the realization of SFE schemes on mobile phones. While our improvements thus far took SFE from impossible to useful for extremely small

problems, we realized that significant changes would need to be made to our approach if we ever hoped to make mobile a practical platform for secure computation. We therefore determined that focusing our efforts on garbled circuits would be necessary, but that our approaches when compared to the work of other groups would need to be fundamentally different. Specifically, while all previous mechanisms assumed that both parties in a two party secure computation are symmetrically provisioned with massive computing resources, we would need to find ways to remove the burden of heavy computation from mobile phones.

This thrust of our work developed mechanisms for the secure outsourcing of SFE computation from constrained devices to more capable infrastructure. Our protocol maintains the privacy of both participant's inputs and outputs while significantly reducing the computation and network overhead required by the mobile device for garbled circuit evaluation. We developed a number of extensions to allow the mobile device to check for malicious behavior from the circuit generator or the cloud and a novel Outsourced Oblivious Transfer for sending garbled input data to the cloud. We then implemented the new protocol on a commodity Android mobile device and reasonably provisioned servers and demonstrate significant performance improvements over evaluating garbled circuits directly on the mobile device.

Our approach is shown in Figure 4. A mobile device (Alice) acts in a modified version of the evaluator role from a traditional garbled circuit protocol. After determining that the generator (Bob, a very well-provisioned server) has properly generated the circuits, Alice performs Outsourced Oblivious Transfers, which deliver her garbled inputs to the Cloud (another well-provisioned server who is also untrusted). The cloud then receives Bob's inputs and the circuits and evaluates the circuits on Alice's behalf. Note that because the Cloud does not know either Alice's or Bob's ungarbled inputs and that Alice has approved the circuits that the Cloud executes, the Cloud learns nothing about the inputs or outputs of the execution of this protocol. Moreover, unlike previous work, the cloud is able to release the output(s) of the computation to both parties simultaneously, reducing the ability of either Alice or Bob to learn the result of a computation without releasing it to the other party.

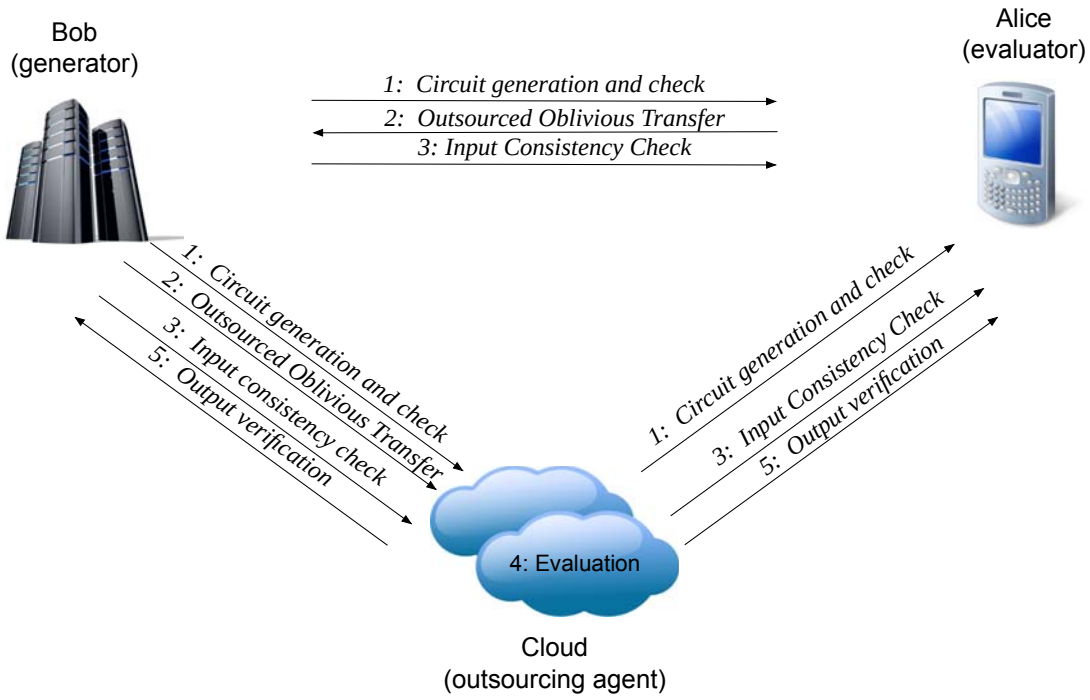


Figure 4: Our "outsourcing" architecture. Here, a cloud helps a mobile device perform the evaluation phase of a garbled circuit protocol without loss of security from the traditional two party model. This approach was able to reduce execution time by over 98%.

Shifting the evaluation phase from a mobile phone to the cloud yielded significant performance improvements. Figure 5 shows the results of a comparison of execution times for the Edit Distance problem for inputs ranging from 2 to 128 bits using our outsourcing approach against the KSS two-party scheme (the fastest two-party scheme at the time of this work). An interesting observation is that the previous largest edit distance circuit executed on a mobile device was of input size 8 using our PAL compiler. Not only did our outsourcing scheme far exceed this, but it was also able to do so 16 times faster than the KSS scheme. Our performance only continues to improve over KSS as we move towards the malicious adversary model. When we compare execution over 32 circuits (thereby reducing the chance an adversary can cheat to $2^{-10.2}$), our execution time over KSS is improved by 98%. Our approach also reduces bandwidth used to communicate with the mobile device by as much as 99.95% (or 1900 times less bandwidth).

This work marked the first time that mobile devices were able to participate in the execution of circuits as large as those being used by their desktop counterparts, but also the first time in which mobile devices were able to participate in protocols secure in the malicious model. PROCEED benchmarks made for all other teams, specifically execution of AES-128, was now possible one mobile devices. However, many improvements remained to be made. Performance in the malicious model, while possible, remained prohibitively expensive for practical usage. Our later work would further refine this model to further reduce the cost of a mobile device participating in such a transaction.

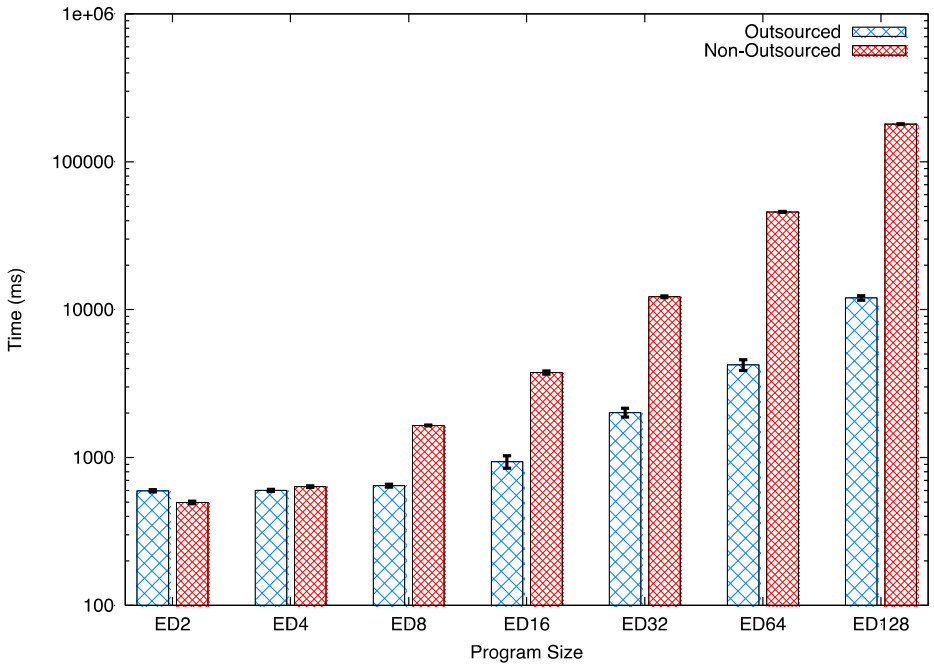


Figure 5: Performance comparison between fastest peer two party computation scheme and our outsourcing approach. Note the log scale on the y-axis.

Portable Circuit Format (PCF)

Our previous work on PAL allowed garbled circuit implementations to gain significant improvements in efficiency through circuit templating. However, as mentioned in that subsection, additional reductions in overhead would need to be made for resource constrained mobile devices. Specifically, compact representations of circuits and functions could dramatically reduce the bandwidth required to transmit garbled circuits between generator and evaluator, minimizing the power and time required to execute such applications.

We refer to our circuit representation as the Portable Circuit Format (PCF). When the SFE system is run, it uses our interpreter to load the PCF program and execute it. As the PCF program runs, it interacts with the SFE system, managing information about gates internally based on the responses from the SFE system itself. In our system, the circuit is ephemeral; it is not necessary to store the entire circuit, and wires will be deleted from memory once they are no longer required. The key insight of our approach is that it is not necessary to unroll loops until the SFE protocol runs. While previous compilers discard the loop structure of the function, ours emits it as part of the control structure of the PCF program. Figure 6 offers a high-level description of our approach.

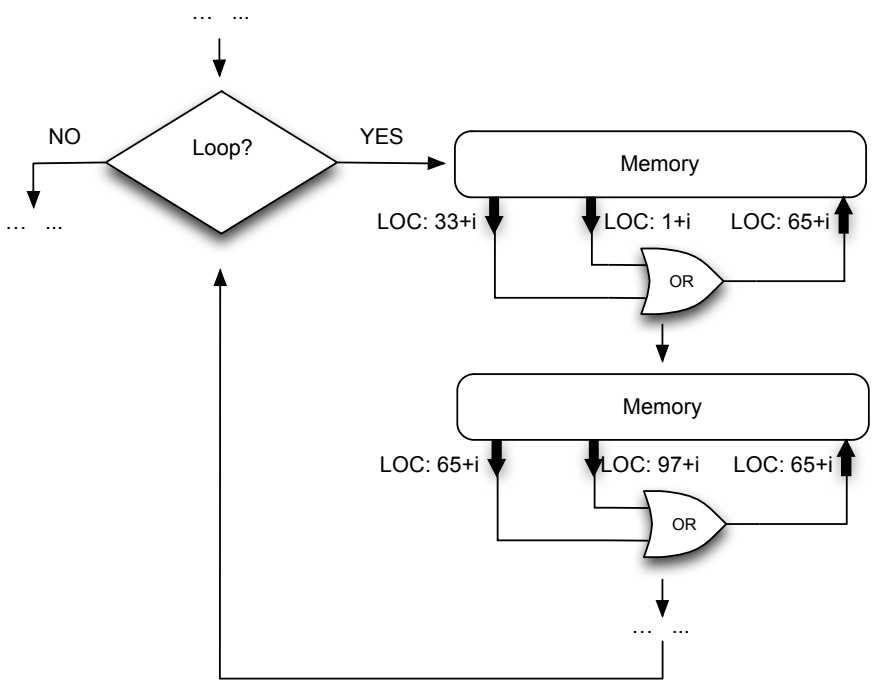


Figure 6: A high-level view of PCF's design. Loops are no longer unrolled at compile time, even to perform optimizations on the circuit. Instead, loops can be evaluated at runtime with gates being computed on-the-fly.

Our system builds upon the PAL and KSS systems to solve the memory scalability problem without sacrificing the ability to optimize circuits automatically. Two observations are key to our approach. One of our most important observations was that it was possible to free the memory required for storing wire values without computing a reference count for the wire. In previous work, each wire in a circuit is assigned a unique global identifier, and gate input wires are specified in terms of these identifiers (output wires can be identified by the position of the gate in the gate list). Rather than using global identifiers, we observe that wire values are ephemeral, and only require a unique identity until their last use as the input to a gate.

These optimizations offered notable improvements in performance over past work. For instance, when compared to the KSS compiler (viewed as the fastest and most efficient at the time of this work), PCF produced circuits only 30% as large from the same source code. With the techniques presented in this work, we also demonstrated that the RSA algorithm with a real-world key size and real-world security level could be compiled and run in a garbled circuit protocol using a typical desktop computer. To the best of our knowledge, the RSA-1024 circuit we tested was larger than any previous garbled circuit experiment, with more than 42 billion gates.

Partially Garbled Circuits and Secure Amortization

Our efforts up to this point made significant improvements in reducing the cost of executing garbled circuit computations against standard metrics – memory utilization, execution time and bandwidth overhead. However, these parameters do not capture every way in which a garbled circuit protocol can be used. One particular example is how often a computation is made, and whether or not accommodations can be made for securely reusing pieces of a transaction in order to amortize costs across protocol iterations. We address this issue through the use of partially garbled circuits, and a system we refer to as PartialGC.

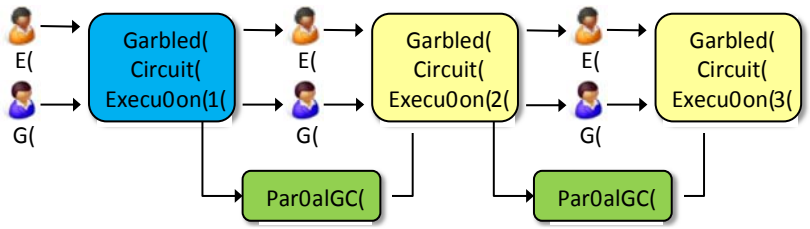


Figure 7: PartialGC overview. The blue box represents a standard evaluation between the (E)valuator and the (G)enerator. Yellow boxes are executions that take partial inputs and produce partial outputs.

Figure 7 presents a high-level overview of the philosophy behind our PartialGC system. First, a standard SFE execution (blue) takes place, at the end of which we "save" some intermediate output values. All further executions use intermediate values from previous executions. In order to reuse these values, information from both parties – the generator and the evaluator – has to be saved. In our protocol, it is the cloud – rather than the evaluator – that saves information. This allows multiple distinct evaluators to participate in a large computation over time by saving state in the cloud between different garbled circuit executions. For example, in a scenario where a mobile phone is outsourcing computation to a cloud, PartialGC can save the encrypted intermediate outputs to the cloud instead of the phone. This allows the devices to communicate with each other by storing encrypted intermediate values in the cloud, which is more efficient than requiring them to directly participate in the saving of values, as required by earlier 2P-SFE systems.

	16 Circuits			64 Circuits			256 Circuits		
	CMTB	PartialGC		CMTB	PartialGC		CMTB	PartialGC	
KeyedDB 64	6.6 ± 4%	1.4 ± 1%	4.7x	27 ± 4%	5.1 ± 2%	5.3x	110 ± 2%	24.9 ± 0.3%	4.4x
KeyedDB 128	13 ± 3%	1.8 ± 2%	7.2x	54 ± 4%	5.8 ± 2%	9.3x	220 ± 5%	27.9 ± 0.5%	7.9x
KeyedDB 256	25 ± 4%	2.5 ± 1%	10x	110 ± 7%	7.3 ± 2%	15x	420 ± 4%	33.5 ± 0.6%	13x
MatrixMult8x8	42 ± 3%	41 ± 4%	1.0x	94 ± 4%	79 ± 3%	1.2x	300 ± 10%	310 ± 1%	0.97x
Edit Distance 128	18 ± 3%	18 ± 3%	1.0x	40 ± 8%	40 ± 6%	1.0x	120 ± 9%	150 ± 3%	0.8x
Millionaires 8192	13 ± 4%	3.2 ± 1%	4.1x	52 ± 3%	8.5 ± 2%	6.1x	220 ± 5%	38.4 ± 0.9%	5.7x

Figure 8: A performance comparison between ParialGC and our original outsourcing scheme (CMTB). Because PartialGC does not have to send entire circuits in each subsequent iteration, it can reduce execution time.

By reducing the amount of information that needs to be transmitted between iterations from entirely new circuits to wire label values, the PartialGC approach dramatically reduces subsequent executions of an application. Figure 8 shows a comparison in execution times against our initial outsourcing efforts. Note that

while the first iteration of the protocol is virtually identical in terms of execution time, execution time is reduced by as much as 10x when portions of the previous execution can be reused. Bandwidth overhead is similarly reduced, with a reduction of as much as 98% in one case.

While PartialGC brings substantial improvements to the execution of garbled circuits on mobile devices, it could potentially benefit from further advances. One particularly important advancement would be an improved base scheme for outsourcing mobile computation.

Whitewash: Outsourcing Garbled Circuit Generation for Mobile Devices

Outsourcing SFE computations from a mobile device to a more powerful cloud has already helped enable dramatic improvements over the direct application of two-party computation schemes. However, a number of challenges remain. Execution in the malicious model, while possible, was still prohibitively expensive for mobile phones. This fact was only exacerbated by the creation of even larger circuits for applications such as RSA. We therefore revisited our outsourcing techniques to ensure that mobile devices could remain part of the larger SFE ecosystem.

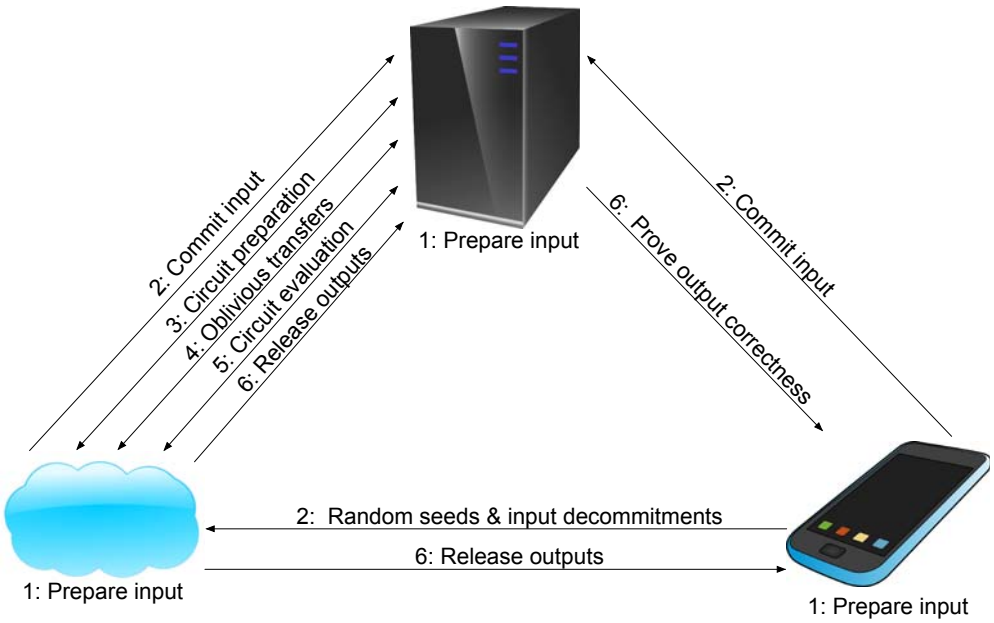


Figure 9: The Whitewash Protocol. Instead of outsourcing evaluation, we outsource circuit generation from the mobile device.

The high-level change between this and our previous work is in which work is outsourced from the mobile. Whereas our previous outsourced circuit evaluation to the Cloud (leaving circuit generation to the other active party), our "Whitewash" technique outsources circuit generation from the mobile device to the Cloud (leaving the other party to perform evaluation). The advantage to this approach is

that it requires a mobile phone to do very little work – simply generating random seeds and committing to its inputs. Outside of these very simple operations, the mobile device simply sits and waits for the two more powerful participants to perform the computation in question. Figure 9 shows the protocol in greater detail.

The improvements from these changes are dramatic. In addition to a performance improvement of as much as 92%, we further reduce network costs by as high as 98%. Additionally, because we incorporated the PCF framework for circuit representation in addition to our protocol improvements, we were able to execute the largest-ever created circuits at the same security levels as traditional two-party.

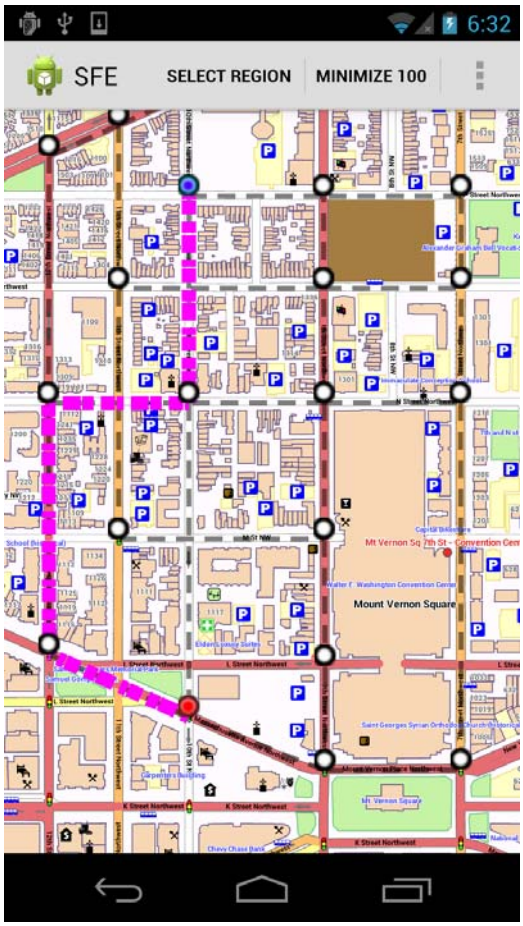


Figure 10: Our privacy-preserving navigation application. A user can learn the most efficient route to their destination without revealing any information about their path.

Figure 10 demonstrates a critical byproduct of the success of our approach, a privacy-preserving navigation application for an Android phone. With this application, a user can query a service for the most efficient route between their current location and their intended destination without revealing such information. Moreover, the mapping service can also help the client make this information without revealing all of its intelligence about the area in question.

Outsourcing Secure Two-Party Computation as a Black Box

While our outsourcing techniques now allow a mobile device to participate in a garbled circuit-based computation as large and secure as a traditional two-party computation, a number of challenges remain. Specifically, while our proofs of security are robust, each of our outsourcing techniques are secure based on the specific sub-components we used at the time. As newer, faster techniques become available, such advances cannot easily be applied to our systems without conducting the expensive process of reproving their security. We would therefore need to develop a means of automatically incorporating potential performance advances.

This effort focused on creating a generic lifting technique for taking any two-party SFE scheme into a secure outsourced scheme. This tradeoff allows for an outsourcing scheme that relies on the underlying two-party protocol in a black-box manner, meaning the underlying protocol can be swapped for any other protocol meeting the same definition of security. Figure 11 presents a high-level overview of our approach. The outsourcing protocol can be informally broken down as follows: first, the mobile device prepares its input by encrypting it and producing a MAC tag for verifying the input is not tampered with before it is entered into the computation. Since the application server and Cloud are assumed not to collude, one party receives the encrypted input, and the other party receives the decryption key. Both of these values are input into the secure two-party computation, and are verified within the secure two-party protocol using the associated MAC tags. If the check fails, the protocol outputs a failure message. Otherwise, the second phase of the protocol, the actual evaluation of the SMC program, takes place. The third and final phase encrypts and outputs the mobile device's result to both parties, who in turn deliver these results back to the mobile device. Intuitively, since our security model assumes that the application server and the Cloud are never simultaneously malicious, at least one of these two will return the correct result to the mobile device. From this, the mobile will detect any tampering from the malicious party by a discrepancy in these returned values, eliminating the need for an output MAC. If no tampering is detected, the mobile device then decrypts the output of computation.

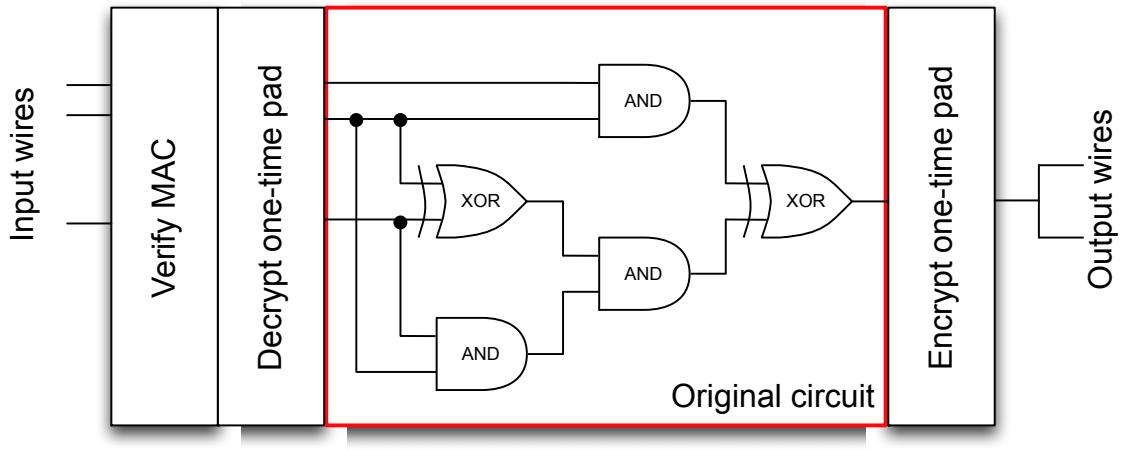


Figure 11: The process of creating a blackbox-ready a circuit. The initial circuit is augmented with a MAC prior to execution and re-encrypted using a one-time pad prior to release.

The performance analysis in this work differs from our previous efforts. Instead of showing the improvement for a mobile device over the direct implementation of a two-party SFE technique, we compared the cost of doing a traditional two-party computation against a blackbox outsourced computation. Our results clearly demonstrate that, as circuits become large, the overhead added by our technique vanishes into the confidence intervals of the two-party protocol execution. Our approach also has similar results for bandwidth overhead. Accordingly, mobile devices will now be able to immediately take advantage of any new two-party scheme with virtually no changes or additional overhead.

We will discuss a number of remaining challenges to outsourcing in the Recommendations Section.

Frigate: A Validated, Extensible, and Efficient Compiler for Secure Computation

At the beginning of this project, the community had few resources it could use to develop real systems based on garbled circuits. The Fairplay compiler created a starting point for a number of other experimental compilers and interpreters, each of which brought increasing efficiency to this field. The research community has now become reliant on these artifacts in order to make the prospect of practical secure computation a reality. Unfortunately as our next work demonstrates, all of the most prominent SFE compilers available at this time contain a significant number of stability and correctness issues, drawing into question the security guarantees they purport.

Throughout the course of the PROCEED project, we gained extensive experience with research artifacts created by a number of different performers. Throughout our interactions with these different tools, we noticed significant issues with each of them. For instance, many compilers failed to generate the correct logic for if

statements, whereas others were brittle and crashed when compiling all but a small number of applications. Figure 12 shows the results of our correctness tests against the five most popular garbled circuit compilers.

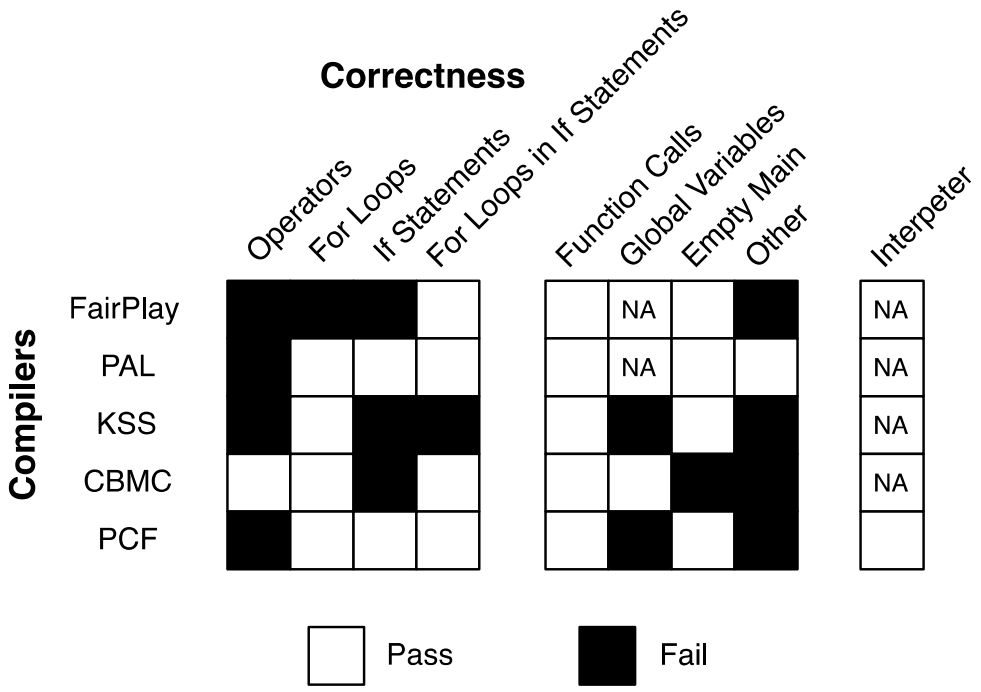


Figure 12: Summary of correctness results. Note that all major SFE compilers currently produce incorrect outputs.

The reasons for these failures became clearer by our extensive work with each of the compilers. Specifically, while the compiler design community has extensive sets of best-practices for designing new compilers, none of these principles appear to have been applied uniformly during the construction of these popular garbled circuit compilers. We addressed this problem directly by designing and implementing the Frigate compiler. We name our compiler after the naval vessel, known for its speed and adaptability for varying missions. Our compiler is designed to be validated through an extensive battery of testing all facets of its operation, modular and extensible to support a variety of research applications, and faster than the state of the art circuit compilers in the community. In addition, the frigate's use as an escort ship parallels the potential for our compiler to facilitate continued secure computation research.

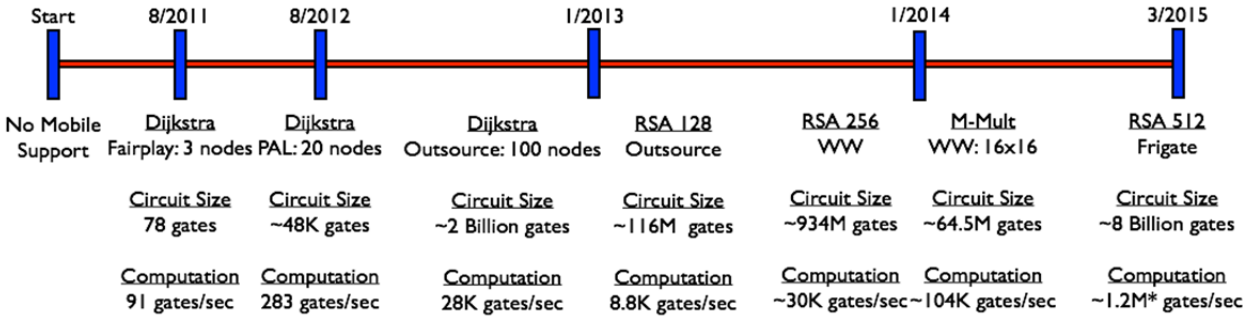
Frigate is made robust by a number of design decisions. First, we rely on standard methodology from the compiler community (e.g., lexing, parsing, semantic analysis and code generation), and use data structures such as abstract syntax trees. Second, we perform extensive compiler output validation. Finally, we provide useful error messages when the compiler detects a problem with an application, making debugging the application significantly easier than with other compilers.

The results of this work provide significant improvements over related efforts. First, our compiler and interpreter were extensively tested using methodology from the compiler community to ensure correct operation. Second, by focusing our new compiler on simple, clean design, we saw a significant improvement in performance. When compared to the current fastest compiler/interpreter pair (KSS), we were able to reduce compilation times by 682x.

Accordingly, this artifact represents the most stable and fastest compiler available to the research community.

Conclusions

At the beginning of this project, the prospect of running flexible and efficient privacy preserving protocols on mobile devices was viewed as unworkable. Our work has made significant progress against this goal, moving a mobile solution from impossible to invisible when compared to traditional two-party computations. Our progress can be seen below in Figure 13.



- ~8 orders of magnitude circuit size
- >4 orders of magnitude performance improvement.
- *Exact same security parameters as best server-class work with indistinguishable performance characteristics.*

Figure 13: Overview of progress during the course of our work on the PROCEED project.

Our mobile solutions are now capable of participating in the execution of the largest-ever generated garbled circuits, at a security level equal to the best traditional two-party schemes with minimal bandwidth overhead.

Recommendations

While our work has made crucial progress in realizing practical privacy-preserving applications on mobile devices, significant advances must still be made in order to widely deploy such systems. We offer the following recommendations to ensuring that these protocols are ultimately realized:

- Fast Navigation: Our work demonstrates that mobile devices can act as first-class participants in secure computation. Our most promising embodiment of this is our privacy-preserving navigation application. However, more work remains in order to move this application from possible to practical. Specifically, an effort to explore faster shortest-path algorithms, more efficient data structures, a reduced adversarial model and map scaling are necessary to deploy this result.

- Improved Compilers: The research community has developed a number of important artifacts throughout the duration of this work. However, many of these research systems are inefficient or suffer from issues of incompleteness. While our effort to create the Frigate compiler is an important first step, the community needs to have a set of tools they can rely upon to make future progress. Accordingly, formal verification (in some form) of a compiler is necessary to continue to move forward.

- Central Project Repository: We have produced a significant amount of code in the process of executing this project, as have a number of other groups. Creating a collection of all software for public use would significantly serve the research community. At the current time, finding working code can be a frustrating and ad hoc process, reducing the time that the community is spending on high quality research.

Appendix

Published Papers (each included)

All papers are Contracted Fundamental Research (CFR) and do not require Pre-Publication Approval

B. Mood, L. Letaw, and K. Butler. **Memory-Efficient Garbled Circuit Generation for Mobile Devices**. 16th IFCA International Conference on Financial Cryptography and Data Security (FC'12). 2012.

H. Carter, B. Mood, P. Traynor, and K. Butler. **Secure Outsourced Garbled Circuit Evaluation for Mobile Devices**. USENIX Security Symposium (Security'13), 2013.

B. Kreuter, a. shelat, B. Mood, and K. Butler. **PCF: A Portable Circuit Format For Scalable Two-Party Secure Computation.** USENIX Security Symposium (Security'13), 2013.

H. Carter, C. Amrutkar, I. Dacosta and P. Traynor, **For Your Phone Only: Custom Protocols for Efficient Secure Function Evaluation on Mobile Devices**, Journal of Security and Communication Networks (SCN), 7(7), p. 1165–1176, 2014.

B. Mood, D. Gupta, K. Butler, and J. Feigenbaum. **Reuse It Or Lose It: More Efficient Secure Computation Through Reuse of Encrypted Values**. ACM Conference on Computer and Communications Security (CCS'14), Scottsdale, AZ, USA, November 2014.

H. Carter, C. Lever, P. Traynor, **Whitewash: Outsourcing Garbled Circuit Generation for Mobile Devices,** Annual Computer Security Applications Conference (ACSAC), December 2014.

H. Carter, B. Mood, K. Butler, P. Traynor, **Outsourcing Secure Two-Party Computation as a Black Box,** In Submission.

B. Mood, D. Gupta, H. Carter, P. Traynor and K. Butler. **Frigate: A Validated, Extensible, and Efficient Compiler for Secure Computation**, In Submission.

Memory-Efficient Garbled Circuit Generation for Mobile Devices

Benjamin Mood, Lara Letaw, and Kevin Butler

Department of Compter & Information Science
University of Oregon, Eugene, OR 97405 USA
{bmood,zephron,butler}@cs.uoregon.edu

Abstract. Secure function evaluation (SFE) on mobile devices, such as smartphones, creates compelling new applications such as privacy-preserving bartering. Generating custom garbled circuits on smartphones, however, is infeasible for all but the most trivial problems due to the high memory overhead incurred. In this paper, we develop a new methodology of generating garbled circuits that is memory-efficient. Using the standard SFDL language for describing secure functions as input, we design a new pseudo-assembly language (PAL) and a template-driven compiler that generates circuits which can be evaluated with Fairplay. We deploy this compiler for Android devices and demonstrate that a large new set of circuits can now be generated on smartphones, with memory overhead for the set intersection problem reduced by 95.6% for the 2-set case. We develop a password vault application to show how runtime generation of circuits can be used in practice. We also show that our circuit generation techniques can be used in conjunction with other SFE optimizations. These results demonstrate the feasibility of generating garbled circuits on mobile devices while maintaining high-level function specification.

1 Introduction

Mobile phones are extraordinarily popular, with adoption rates unprecedented in the history of product adoption by consumers. Smartphones in particular have been embraced, with over 296 million of these devices shipped in 2010 [4]. The increasing importance of the mobile computing environment requires functionality tailored to the limited resources available on a phone. Concerns of portability and battery life necessitate design compromises for mobile devices compared to servers, desktops, and even laptops. In short, mobile devices will always be resource-constrained compared to their larger counterparts. However, through careful design and implementation, they can provide equivalent functionality while retaining the advantages of ubiquitous access.

Privacy-preserving computing is particularly well suited to deployment on mobile devices. For example, two parties bartering in a marketplace may wish to conveal the nature of their transaction from others, and share minimal information with each other. Such a transaction is ideally suited for *secure function evaluation*, or SFE. Recent work, such as by Chapman et al. [6], demonstrates the myriad applications of SFE on smartphones.

However, because of computational and memory requirements, performing many of these operations in the mobile environment is infeasible; often, the only hope is outsourcing computation to a cloud or other trusted third party, thus raising concerns about the privacy of the computation.

In this paper, we describe a memory-efficient technique for generating the garbled circuits needed to perform secure function evaluation on smartphones. While numerous research initiatives have considered how to *evaluate* these circuits more efficiently [16, 7], little work has gone towards efficient *generation*. Our port of the canonical Fairplay [12] compiler for SFE to the Android mobile operating system revealed that because of intensive memory requirements, the majority of circuits could not be compiled in this environment. As a result, our main contribution is a novel design to compile the high-level Secure Function Definition Language (SFDL) used by Fairplay and other SFE environments into garbled circuits with minimal memory usage. We created Pseudo Assembly Language (PAL), a mid-level intermediate representation (IR) compiled from SFDL, where each instruction represents a pre-built circuit. We created a Pseudo Assembly Language Compiler (PALC), which takes in a PAL file and outputs the corresponding circuit in Fairplay's syntax. We then created a compiler to compile SFDL files into PAL and then, using PALC, to the Secure Hardware Definition Language (SHDL) used by Fairplay for circuit evaluation.

Using these compilation techniques, we are able to generate circuits that were previously infeasible to create in the mobile environment. For example, the set intersection problem with sets of size two requires 469 KB of memory with our techniques versus over 10667 KB using a direct port of Fairplay to Android, a reduction of 95.6%. We are able to evaluate results for the set intersection problem using four and eight sets, as well as other problems such as Levenshtein distance; none of these circuits could previously be generated at all on mobile devices due to their memory overhead. Combined with more efficient evaluation, our techniques provide a new arsenal for making privacy-preserving computation feasible in the mobile environment.

The rest of this paper is organized as follows. Section 2 provides background on secure function evaluation, garbled circuits, and the Fairplay SFE compiler. Section 3 describes the design of PAL, our pseudo assembly language, and our associated compilers. Section 4 describes our testing environment and methodology, and provides benchmarks on memory and execution time. Section 5 describes applications that demonstrate circuit generation in use, while Section 6 describes related work and Section 7 concludes.

2 Background

2.1 Secure Function Evaluation with Fairplay

The origins of SFE trace back to Yao's pioneering work on garbled circuits [18]. SFE enables two parties to compute a function without knowing each other's input and without the presence of a trusted third party. More formally, given

participants Alice and Bob with input vectors $\boldsymbol{a} = a_0, a_1, \cdots a_{n-1}$ and $\boldsymbol{b} = b_0, b_1, \cdots b_{m-1}$ respectively, they wish to compute a function $f(\boldsymbol{a}, \boldsymbol{b})$ without revealing any information about the inputs that cannot be gleaned from observing the function's output. Fundamentally, SFE is predicated on two cryptographic primitives. *Garbled circuits* allow for the evaluation of a function without any party gaining additional information about the participants. This is possible since one party creates a garbled circuit and the other party evaluates the circuit without knowing what the wires represent. Secondly, *oblivious transfer* allows the party executing the garbled circuit to obtain the correct wires for setting inputs from the other party without gaining additional information about the circuit; in particular, a 1-out-of-n OT protocol allows Bob to learn about one piece of data without gaining any information on the remaining $n - 1$ pieces.

A garbled circuit is composed of many garbled gates, with inputs represented by two random fixed-length strings. Like a normal boolean gate, the garbled gate evaluates the inputs and gives a single output, but alterations are made to the garbled gate's truth table: aside from the randomly chosen input values, the output values are uniquely encrypted by the input wires and an initialization vector. The order of the entries in the table is then permuted to prevent the order from giving away the value. Consequently, the only values saved for the truth table are the four encrypted output values. A two-input gate is thus represented by the two inputs and four encrypted output values.

The garbled circuit protocol requires that both parties are able to provide inputs. If Bob creates the circuit and Alice receives it, Bob can determine which wires to set, and Alice performs an oblivious transfer to receive her input wires. Once she knows her input wires she runs the circuit by evaluating each gate in order. To evaluate a gate, she uses the input values as the key to decrypt the output value. To find the correct entry in the table, Alice uses a decryption step using the input wires as keys. To find her output, Alice acquires a translation table, a hash of the wires, from Bob for her possible output values. She then can perform the hash on her output wires to see which wires were set. Alice sends Bob's output in garbled form since she cannot interpret it.

Fairplay is the canonical tool for generating and evaluating garbled circuits for secure function evaluation. The Fairplay group is notable for creating the abstraction of a high-level language, known as SFDL. This language describes secure evaluation functions and is compiled SHDL, which is written in the style of a hardware description language such as VHDL and describes the garbled circuit. The circuit evaluation portion of Fairplay provides for the execution of the garbled circuit protocol and uses oblivious transfer (OT) to exchange information. Fairplay uses the 1-out-of-2 OT protocols of Bellare et al. [1] and Naor et al. [14] which allows for Alice to pick one of two items that Bob is offering and also prevents Bob from knowing which item she has picked.

Examining the compiler in more detail, Fairplay compiles each instruction written in SFDL into a so-called *multi-bit instruction*. These multi-bit (e.g. integer) instructions are transformed to *single-bit instructions* (e.g., the 32 separate bits to represent that integer). From these single-bit instructions, Fairplay then

(a) Fairplay compiler process. (b) PAL compiler process.

Fig. 1: Compilation with Fairplay versus PAL.

unrolls variables, transforms the instructions into SHDL, and outputs the file, either immediately or after further circuit optimizations.

Fairplay's circuit generation process is very memory-intensive. We performed a port of Fairplay directly to the Android mobile platform (described further in Section 4) and found that a large number of circuits were completely unable to be compiled. We examined the results of circuit compilation on a PC to determine the scope of memory requirements. From tests we performed on a 64-bit Windows 7 machine, we observed that Fairplay needed at least 245 MB of memory to run the compilation of the keyed database program, a program that matches database keys with values and employs SFE for privacy preservation (described further in Section 4). In order to determine the cause of this memory usage, we began by analyzing Fairplay's compiler.

From our analysis, Fairplay uses the most memory during the mapping operation from multi-bit to single-bit instructions. During this phase, the memory requirements increased by 7 times when the keyed database program ran. We concluded that it would be easier to create a new system for generating the SHDL circuit file, rather than making extensive modifications to the existing Fairplay implementation. To accomplish this, we created an intermediate language that we called PAL, described in detail in section 3.

2.2 Threat Model

As with Fairplay, which is secure in the random oracle model implemented using the SHA-1 hash function, our threat model accounts for an honest-but-curious adversary. This means the participants will obey the given protocol but may look at any data the protocol produces. Note that this assumption is well-described by others considering secure function and secure multiparty computation, such as Kruger et al.'s OBDD protocol [10], Pinkas et al.'s SFE optimizations [16], the TASTY proposal for automating two-party communication [5], Jha et al.'s privacy-preserving genomics [8], Brickell et al.'s privacy-preserving classifiers [3] and Huang et al.'s recent improvements to evaluating SFE [6]. Similarly, we make the well-used assumption that parties enter correct input to the function.

3 Design

To overcome the intensive memory requirements of generating garbled circuits within Fairplay, we designed a *pseudo assembly language*, or PAL, and a *pseudo*

Possible Operations	
Operation	Syntax
Addition	DEST + V1 V2
Greater than or Equal to	DEST >= V1 V2
Equal to	DEST == V1 V2
Bitwise AND	DEST & V1 V2
If Conditional	DEST IF COND V1 V2
Input line	INPUT V1 a (or INPUT V1 b)
Output line	INPUT V1 a (or INPUT V1 b)
For loop	V1 FOR X (an integer) to Y (an integer)
Call a procedure	V1 PROC
Call a function	DEST,...,DEST = FunctionName(param, ... ,param)
Multiple Set Equals	DEST,...,DEST=V,...,V

Table 1: PAL Operations

assembly language compiler called PALC. As noted in Figure 1, we change Fairplay's compilation model by first compiling SFDL files into PAL using our FP-PALC compiler, and generating the SHDL file which can then be run using Fairplay's circuit evaluator with our PALC compiler.

3.1 PAL

We first describe PAL, our memory-efficient language for garbled circuit creation. PAL resembles an assembly language where each instruction corresponds to a pre-optimized circuit. PAL is composed of at least two parts: variable declarations and instructions. PAL files may also contain functions and procedures. A full table showing all headings can be found in the full technical report [13] and is elided here because of space constraints.

Table 1 lists an abbreviated set of operations that are available in PAL along with their instruction signatures. The full set can be found in our technical report [13]. Each operation consists of a destination, an operator, and one to three operands. `DEST`, `V1`, `V2`, and `COND` are variables in our operation listing. PAL also has operations not found in Fairplay, such as shift and rotate.

Note that conditionals can be reduced to the `IF` conditional. Unlike in regular programs, all parts of an `IF` circuit must be executed on every run.

The first part of a PAL program is the set of variable declarations. These consist of a variable name and bit length, and the section is marked by a *Variables:* label. In this low-level language there are no structs or objects, only integer variables and arrays. Each variable in a PAL file must be declared before it can be used. Array indices may be declared at any point in the variable name.

Figure 2 shows an example of variables declared in PAL. `Alicekey` and `Bobkey` have a bit length of 6, `Bobin` and `Aliceout` have a bit length of 32, `COND` is a boolean like variable which has a bit length of 1, and `Array[7]` is an array of seven elements where each have a bit length of 5. All declared variables

```
Variables:
Alicekey    6
Bobin       32
Bobkey      6
Aliceout    32
COND        1
Array[7]    5
```

Fig. 2: Example of variable declarations in PAL.

```
Instructions:
Bobin IN b
Bobkey IN b
Alicekey IN a
COND == Alicekey Bobkey
Aliceout IF COND Bobin Aliceout
Aliceout OUT a
```

Fig. 3: Example of number comparison (for keyed database problem) in PAL.

```
Variables:
i 6
in.a 6
in.b[16].data 24
in.b[16].key 6
out.a 24
$c0 1
$t0 1
DBsize 64

Procedure: $p0
$t0 == in.a in.b[i].key
$c0 = $t0
out.a IF $c0 in.b[i].data out.a

Instructions:
in.b[16].data IN b
in.b[16].key IN b
in.a IN a
DBsize = 16
i FOR 0 15
$p0 PROC
out.a OUT a
```

Fig. 4: Representation of keyed database program in PAL.

are initialized to 0. After variable declarations, a PAL program can have function and procedure definitions preceding the instructions, which is the main function.

Figure 3 shows the PAL instructions for comparing two keys as used in the keyed database problem, described more fully below. The first two statements are input retrieval for Bob, while the third retrieves input for Alice. A boolean like variable `COND` is set based on a comparison and the output is set accordingly. Note that constants are allowed in place of `V1`, `V2`, or `COND` in any instruction. PAL supports loops, functions, and procedures.

To illustrate a full program, Figure 4 shows the keyed database problem in PAL, where a user selects data from another user's database without any information given about the item selected. In this program, Bob enters 16 keys and 16 data entries and Alice enters her key. If Alice's key matches one of Bob's then Alice's output of the program is Bob's data entry that held the corresponding key. The PAL program shows how each key is checked against Alice's key. If one of those keys matches, then the output is set.

3.2 PALC

Circuits generated by our PALC compiler, which generates SHDL files from PAL, are created using a database of pre-generated circuits matching instructions to

their circuit representations. These circuits, other than equality, were generated using simple Fairplay programs that represent equivalent functionality. Any operation that does not generate a gate is considered a *free* operation. Assignments, shifts, and rotates are free.

Variables in PALC have two possible states: they are either specified by a list of gate positions or they have a real numerical value. If an operation is performed on real value variables, the result is stored as a real value. These real value operations do not need a circuit to be created and are thus free.

When variables of two different sizes are used, the size of the operation is determined by the destination. If the destination is 24 bits and the operands are 32 bits, the operation will be performed 24-bit operands. This will not cause an error but may yield incorrect results if false assumptions are made.

There are currently a number of known optimizations, such as removing static gates, which are not implemented inside PALC; these optimization techniques are a subject of future work.

3.3 FPPALC

To demonstrate the feasibility of compileing non-trivial programs on a phone, we modified Fairplay's SFDL compiler to compile into PAL and then run PALC to compile to SHDL. This compiler is called FPPALC. Compiling in steps greatly reduces the amount of memory that is required for circuit generation.

We note our compiler will not yield the same result as Fairplay's compiler in two cases, which we believe demonstrate erroneous behavior in Fairplay. In these instances, Fairplay's circuit evaluator will crash or yield erroneous results. A more detailed explanation can be found in our technical report [13], To summarize, unoptimized constants in SFDL can cause the evaluator to crash, while programs consisting of a single `if` statement can produce inconsistent variable modifications. Apart from these differences, the generated circuits have equivalent functionality.

For our implementation of the SFDL to PAL compiler we took the original Fairplay compiler and modified it to produce the PAL output by removing all elements besides the parser. From the parser we built our own type system, support for basic expressions, assignment statements, and finally `if` statements and `for` loops. All variables are represented as unsigned variables in the output but input and other operations treat them as signed variables. Our implementation of FPPALC and PALC, which compile SFDL to PAL and PAL to SHDL respectively, comprises over 7500 lines of Java code.

3.4 Garbled Circuit Security

A major question posed about our work is the following: *Does using an intermediate metalanguage with precompiled circuit templates change the security guarantees compared to circuits generated completely within Fairplay?* The simple answer to this question is no: we believe that the security guarantees offered by the circuits that we compile with PAL are equivalent to those from Fairplay.

	Memory (KB)			Time (ms)		
Program	Initial	SFDL→PAL	PAL→SHDL	SFDL→PAL	PAL→SHDL	Total
Millionaires	4931	5200	5227	90	29	119
Billionaires	4924	5214	5365	152	54	206
CoinFlip	5042	5379	5426	139	122	261
KeyedDB	4971	5365	5659	142	220	362
SetInter 2	5064	5393	5533	161	305	466
SetInter 4	5078	5437	5600	135	1074	1209
SetInter 8	5122	5542	5739	170	6659	6829
Levenshtein Dist 2	5184	5431	5576	183	336	519
Levenshtein Dist 4	5233	5436	5638	190	622	802
Levenshtein Dist 8	5264	5473	5693	189	2987	3172

Table 2: FPPALC on Android: total memory application was using at end of stages and the time it took.

Because there are no preconditions about the design of the circuit in the description of our garbled circuit protocol, any circuit that generates a given result will work: there are often multiple ways of building a circuit with equivalent functionality. Additionally, the circuit construction is a composition of existing circuit templates that were themselves generated through Fairplay-like constructions. Note that the security of Fairplay does not rely on how the circuits are created but on the way garbled circuit constructs work. Therefore, our circuits will provide the same security guarantees since our circuits also rely on using the garbled circuit protocol.

4 Evaluation

In this section, we demonstrate the performance of our circuit generator to show its feasibility for use on mobile devices. We targeted the Android platform for our implementation, with HTC Thunderbolts as a deployment platform. These smartphones contain a 1 GHz Qualcomm Snapdragon processor and 768 MB of RAM, with each Android application limited to a 24 MB heap.

4.1 Testing Methodology

We benchmarked compile-time resource usage with and without intermediate compilation to the PAL language. We tested on the Thunderbolts; all results reported are from these devices. Memory usage on the phones was measured by looking at the PSS metric, which measures pages that have memory from multiple processes. The PSS metric is an approximation of the number of pages used combined with how many processes are using a specific page of memory.

Several SFDL programs, of varying complexity, were used for benchmarking. Each program is described below. We use the SFDL programs representing the

Millionaires, Billionaires, and Keyed Database problems as presented in Fairplay [11]. The other SFDL files that we have written can be found in the full technical report [13]. We describe these below in more detail.

The *Millionaire's* problem describes two users who want to determine which has more money without either revealing their inputs. We used a 4-bit integer input for this problem. The *Billionaire's* problem is identical in structure but uses 32-bit inputs instead. The *CoinFlip* problem models a trusted coin flip where neither party can determine the program's outcome deterministically. It takes two inputs of 24-bit inputs per party. In the *Keyed database* program, a user performs a lookup in another user's database and returns a value without the owner being aware of which part of the database is looked up – we use a database of size 16. The keys are 6-bits and the data members are 24-bits. The *Set intersection* problem determines elements two users have in common, e.g., friends in a social network. We measured with sets of size 2, 4, and 8 where 24-bit input was used. Finally, we examined *Levenshtein distance*, which measures edit distance between two strings. This program takes in 8-bit inputs.

4.2 Results

Below the results of the compile-time tests performed on the HTC Thunderbolts. We measured memory allocation and time required to compile, for both the Fairplay and PAL compilers. In the latter case, we have data for compiling to and from the PAL language. Our complete compiler is referred to FPPALC in this section.

Memory Usage & Compilation Time Table 2 provides memory and execution benchmarks for circuit generation, taken over at least 10 trials per circuit. We measure the initial amount of memory used by the application as an SFDL file is loaded, the amount of memory consumed during the SFDL to PAL compilation, and memory consumed at the end of the PAL to SHDL compilation.

As an example of the advantages of our approach, we successfully compiled a set intersection of size 90 that had 33,000,000 gates on the phone. The output file was greater than 2.5 GB. Android has a limit of 4 GB per file and if this was not the case we believe we could have compiled a file of the size of the memory card (30 GB). This is because the operations are serialized and the circuit never has to fully remain in memory.

Although we did not focus on speed, Table 2 gives a clear indication of where the most time is used per compilation: the PAL to SHDL phase, where the circuit is output. The speed of this phase is directly related to the size of the program that is being output, while the speed of the SFDL to PAL compliation is related to the number of individual instructions.

Comparison to Fairplay Table 3 compares the Fairplay compiler with FPPALC. Where results are not present for Fairplay are situations where it was unable to compile these programs on the phone. For the set intersection problem

	Memory (KB)	
Program	Fairplay	FPPALC
Millionaires	658	296
Billionaires	1188	441
CoinFlip	1488	384
KeyedDB 16	NA	688
SetInter 2	10667	469
SetInter 4	NA	522
SetInter 8	NA	617
Levenshtein Dist 2	NA	392
Levenshtein Dist 4	NA	405
Levenshtein Dist 8	NA	429

Table 3: Comparison of memory increase by Fairplay and FPPALC during circuit generation.

	Memory (KB)			Time (ms)		
Program	Initial	Open File	End	Open File	Fairplay	Nipane
Millionaires	5466	5556	5952	197	533	406
Billionaires	5451	5894	6287	579	1291	981
CoinFlip	5461	5933	6426	789	1795	1320
KeyedDB 16	5315	6197	7667	1600	1678	1593
SetInter 2	5423	5993	6932	1511	2088	1719
SetInter 4	5414	7435	11711	8619	7714	7146
Levenshtein Dist 2	5617	6134	7162	1799	2220	2004
Levenshtein Dist 4	5615	7215	10787	7448	6538	6150
Levenshtein Dist 8	5537	12209	20162	29230	29373	27925

Table 4: Evaluating FPPALC circuits on Fairplay's evaluator with both Nipane et al.'s OT and the suggested Fairplay OT.

with set 2, FPPALC uses 469 KB of memory versus 10667 KB by Fairplay, a reduction of 95.6%. Testing showed that the largest version of the keyed database problem that Fairplay could handle is with a database of size 10, while we easily compiled the circuit with a database of size 16 using FPPALC.

Circuit Evaluation Table 4 depicts the memory and time of the evaluator running the programs compiled by FPPALC. Consider again the two parties Bob and Alice, who create and receive the circuit respectively in the garbled circuit protocol. This table is from Bob's perspective, who has a slightly higher memory usage and a slightly lower run time than Alice. We present the time required to open the circuit file for evaluation and to perform the evaluation using two different oblivious transfer protocols. Described further below, we used both Fairplay's evaluator and an improved oblivious transfer (OT) protocol developed

	Memory (KB)			Time (ms)	
Program	Initial	After File Opening	End	File Opening	Evaluating
Millionaires	5640	5733	5995	194	302
Billionaires	5536	5885	6303	631	958
+CoinFlip	5528	5796	6280	428	1062
KeyedDB 16	5551	6255	7848	2252	1955
SetInter 2	5439	6018	7047	1663	2131
SetInter 4	5553	7708	13507	10540	9555
+Levenshtein Dist 2	5568	5872	6316	529	781
+Levenshtein Dist 4	5577	6088	7178	1704	2213
Levenshtein Dist 8	5488	7670	13011	9745	8662

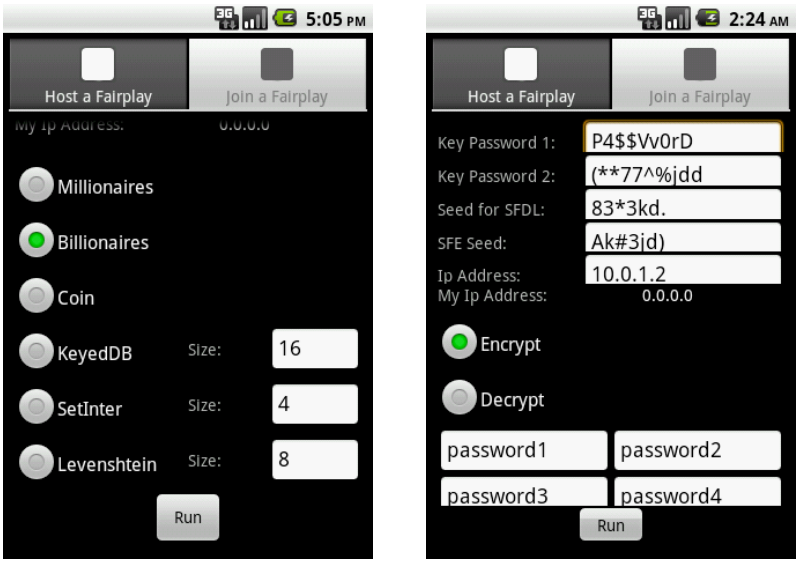
Table 5: Results from programs compiled with Fairplay on a PC evaluated with Nipane et al.'s OT.

by Nipane et al. [15]. Note that Fairplay's evaluator was unable to evaluate programs with around 20,000 mixed two and three input gates on the phone. This limit translates to 209 32-bit addition operations in our compiler.

While the circuits we generate are not optimized in the same manner as Fairplay's circuits, we wanted to ensure that their execution time would still be competitive against circuits generated by Fairplay. Because of the limits of generating Fairplay circuits on the phone, we compiled them using Fairplay on a PC, then used these circuits to compare evaluation times on the phone. Table 5 shows the results of this evaluation. Programs denoted with a + required edits to the SHDL to run in the evaluator, in order to prevent their crashing due to the issues described in Section 3.3. In many cases, evaluating the circuit generated by FPPALC resulted in faster evaluation. One anomaly to this trend was Levenshtein distance, which ran about three times slower using FPPALC. We speculate this is due to the optimization of constant addition operations and discuss further in Section 5. Note, however, that these circuits are unable to be generated on the phone using Fairplay and require pre-compilation.

4.3 Interoperability

To show that our circuit generation protocol can be easily used with other improved approaches to SFE, we used the faster oblivious transfer protocol of Nipane et al. [15], who replace the OT operation in Fairplay with 1-out-of-2 OT scheme based on a two-lock RSA cryptosystem. Shown in Table 5, these provide an over 24% speedup for the Billionaire's problem and 26% speedup for the Coin Flip protocol. On average, there was an 13% decrease in evaluation time across all problems. For the *Millionaires*, *Billionaires*, and *CoinFlip* programs we disabled Nagle's algorithm as described by Nipane et al., leading to better performance on these problems. The magnitude of improvement decreased as circuits increased in size, a situation we continue to investigate. Our main find-



(a) (b)

Fig. 5: Screenshots of editor and password wallet applications.

ings, however, are that our memory-efficient circuit generation is complementary to other approaches that focus on improving execution time and can be easily integrated.

5 Discussion

To demonstrate how our memory-efficient compiler can be used in practice, we developed Android apps capable of generating circuits at runtime. We describe these below.

5.1 GUI Based Editor

To allow compilation on a phone we have to address one large problem. Our experience porting Fairplay to Android showed the difficulty of writing a program on the phone. Figure 5 (a) shows an example of a GUI front-end for picking and compiling given programs based on parameters. A list of programs is given to the user who can then pick and choose which program they wish to run. For some of the programs there is a size variable that can also be changed.

5.2 Password Vault Application

We designed an Android application that introduces SFE as a mechanism to provide secure digital deposit boxes for passwords. In brief, this "password vault"

can work in a decentralized fashion without reliance on the cloud or any third parties. If Alice fears that her phone may go missing and wants Bob to have a copy of her passwords, she and Bob can use their "master" passwords, along with a seed value, as input to a pseudorandomly generated hash function. These master inputs are not revealed to either party, nor is the output of the hash, which is used to encrypt the password. If the passwords are ever lost, Alice can call Bob and jointly recover the passwords; both must present their master passwords to decrypt the password file, ensuring that neither can be individually coerced to retrieve the contents. Figure 5 (b) shows a screenshot of this application. which can encrypt passwords from the user or decrypt those in the database. Our evaluation shows that compiling the hash function requires 6407 KB of memory and approximately 7348 ms, with 85% of that time is the PAL to SHDL conversion. Evaluating the circuit is more time intensive. Opening the file takes 28.1 seconds, and performing the OTs and gate evaluation takes 23.2 seconds. We are exploring efficiencies to reduce execution time.

5.3 Experiences with Garbled Circuit Generation

One of the most important lessons from our implementation efforts was observing the large burden on mobile devices caused when complete circuits must be kept in memory. Better solutions only use small amounts of memory to direct the actual computation, for instance, one copy of each circuit instead of N for N of the same type of statement.

The largest difficulty of the full circuit approach is the need for the full circuit to be created. Circuits for $O(n^2)$ algorithms and beyond scale extremely poorly. A different approach is needed for larger scalabiltiy. For instance, doubling the Levenshtien distance n paremeter increased the circuit size by a factor of about 4.5 (decreasing the larger n grows), when n is 8 there are 11,268 gates, 16 is 51,348 gates, 32 is 218,676 gates, and 64 is 902,004 gates.

The original PAL did not scale well due to the fact it did not have loops, arrays, procedures, or functions. Once those programming structures were added the length of the PAL files were decreased dramaticly. Instead of unrolling all programming control flow constructs we added them for smaller PAL programs. The resulting circuits generated from the new PAL were very similar to the original circuits.

6 Related work

Other research has primarily focused on optimizing the actual evaluation for SFE, while we focus on generating circuits in a memory efficient manor. Kolesnikov et al. [9] demonstrated a "free XOR" evaluation technique to improve execution speed, while Pinkas et al. [16] implement techniques to reduce circuit size of the circuits and computation length. We plan to implement these enhancements in the next version of the circuit evaluator.

Huang et al. [7] have similarly focused on optimizing secure function evaluation, focusing on execution in resource-constrained environments. The approach differs considerably from ours in that users build their own functions directly at the circuit level rather than using high-level abstractions such as SFDL. While the resulting circuit may execute more quickly, there is a burden on the user to correctly generate these circuits, and because input files are generated at the circuit level in Java, compiling on the phone would require a full-scale Java compiler rather than the smaller-scale SFDL compiler that we use.

Another way to increase the speed of SFE has been to focus on leveraging the hardware of devices. Pu et al. [17] have considered leveraging Nvidia's CUDA-based GPU architecture to increase the speed of SFE. We have conducted preliminary investigations into leveraging vector processing capabilities on smartphones, specifically single-instruction multiple-data units available on the ARM Cortex processing cores found within many modern smartphones, as a means of providing better service for certain cryptographic functionality.

Kruger et al. [10] described a way to use ordered binary decision diagrams (OBDDs) to evaluate SFE, which can provide faster execution for certain problems. Our future work will involve determining whether the process of preparing OBDDs can benefit from our memory-efficient techniques. TASTY [5] also uses different methods of privacy-preserving computation, namely homomorphic encryption (HE) as well as garbled circuits, based on user choices. This approach requires the user to explicitly choose the computation style, but may also benefit from our generation techniques for both circuits and the homomorphic constructions. FairplayMP [2] showed a method of secure multiparty computation. We are examining how to extend our compiler to become multiparty capable.

7 Conclusion

We introduced a memory efficient technique for making SFE tractable on the mobile platform. We created PAL, an intermediate language, between SFDL and SHDL programs and showed that by using pre-generated circuit templates we could make previously intractable circuits compile on a smartphone, reducing memory requirements for the set intersection circuit by 95.6%. We demonstrate the use of this compiler with a GUI editor and a password vault application. Future work includes incorporating optimizations in the circuit evaluator and determining whether the pre-generated templates may work with other approaches to both SFE and other privacy-preserving computation primitives.

Acknowledgements

We would like to thank Patrick Traynor for his insights regarding the narrative of the paper, and Adam Bates and Hannah Pruse for their comments.

This material is based on research sponsored by DARPA under agreement number FA8750-11-2-0211. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright

notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

1. M. Bellare and S. Micali. Non-Interactive Oblivious Transfer and Applications. In *International Crytology Conference*, 1990.
2. A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a System for Secure Multi-Party Computation. In *15th ACM Conference on Computer and Communications Security (CCS'08)*, Alexandria, VA, 2008.
3. J. Brickell and V. Shmatikov. Privacy-Preserving Classifier Learning. In *Proceedings of Financial Cryptography and Data Security*, Feb. 2009.
4. Gartner. Gartner Says Worldwide Mobile Device Sales to End Users Reached 1.6 Billion Units in 2010; Smartphone Sales Grew 72 Percent in 2010. http://www.gartner.com/it/page.jsp?id=1543014, 2011.
5. W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-Party Computations. In *17th ACM Conf. on Computer and communications security (CCS'10)*, Chicago, IL, Oct. 2010.
6. Y. Huang, P. Chapman, and D. Evans. Privacy-Preserving applications on smartphones: Challenges and opportunities. In *USENIX HotSec*, Aug. 2011.
7. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *20th USENIX Security Symposium*, Aug. 2011.
8. S. Jha, L. Kruger, and V. Shmatikov. Towards Practical Privacy for Genomic Computation. In *2008 IEEE Symp. on Security and Privacy*, Nov. 2008.
9. V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of ICALP '08*, Reykjavik, Iceland, 2008.
10. L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure Function Evaluation with Ordered Binary Decision Diagrams. In *13th ACM conference on Computer and communications security (CCS'06)*, Alexandria, VA, Oct. 2006.
11. D. Malkhi, N. Nisan, and B. Pinkas. Fairplay Project, http://www.cs.huji.ac.il/project/Fairplay/.
12. D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: a Secure Two-Party Computation System. In *13th USENIX Security Symposium*, San Diego, CA, 2004.
13. B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. Technical Report CIS-TR-2011-04, Department of Computer and Information Science, University of Oregon, Eugene, OR, USA, Sept. 2011.
14. M. Naor and B. Pinkas. Efficient Oblivious Transfer Protocols. In *Proceedings of SODA '01*, Washington, DC, 2001.
15. N. Nipane, I. Dacosta, and P. Traynor. "Mix-In-Place" Anonymous Networking Using Secure Function Evaluation. In *Proceedings of ACSAC*, Dec. 2011.
16. B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation Is Practical. In *Proceedings of ASIACRYPT*, Tokyo, Japan, 2009.
17. S. Pu, P. Duan, and J.-C. Liu. Fastplay–A Parallelization Model and Implementation of SMC on CUDA based GPU Cluster Architecture. Cryptology ePrint Archive, Report 2011/097, 2011. http://eprint.iacr.org/.
18. A. C.-C. Yao. How to Generate and Exchange Secrets. In *Proceedings of the 27th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society.

Secure Outsourced Garbled Circuit Evaluation for Mobile Devices

Henry Carter
Georgia Institute of Technology
carterh@gatech.edu

Benjamin Mood
University of Oregon
bmood@cs.uoregon.edu

Patrick Traynor
Georgia Institute of Technology
traynor@cc.gatech.edu

Kevin Butler
University of Oregon
butler@cs.uoregon.edu

Abstract

Garbled circuits provide a powerful tool for jointly evaluating functions while preserving the privacy of each user's inputs. While recent research has made the use of this primitive more practical, such solutions generally assume that participants are symmetrically provisioned with massive computing resources. In reality, most people on the planet only have access to the comparatively sparse computational resources associated with their mobile phones, and those willing and able to pay for access to public cloud computing infrastructure cannot be assured that their data will remain unexposed. We address this problem by creating a new SFE protocol that allows mobile devices to securely outsource the majority of computation required to evaluate a garbled circuit. Our protocol, which builds on the most efficient garbled circuit evaluation techniques, includes a new outsourced oblivious transfer primitive that requires significantly less bandwidth and computation than standard OT primitives and outsourced input validation techniques that force the cloud to prove that it is executing all protocols correctly. After showing that our extensions are secure in the malicious model, we conduct an extensive performance evaluation for a number of standard SFE test applications as well as a privacy-preserving navigation application designed specifically for the mobile use-case. Our system reduces execution time by 98.92% and bandwidth by 99.95% for the edit distance problem of size 128 compared to non-outsourced evaluation. These results show that even the least capable devices are capable of evaluating some of the largest garbled circuits generated for any platform.

1 Introduction

Secure Function Evaluation (SFE) allows two parties to compute the result of a function without either side having to expose their potentially sensitive inputs to the other. While considered a generally theoretical curiosity even after the discovery of Yao's garbled circuit [43], recent advances in this space have made such computation increasingly practical. Today, functions as complex as AES-128 and approaching one billion gates in size are possible at reasonable throughputs, even in the presence of a malicious adversary.

While recent research has made the constructions in this space appreciably more performant, the majority of related work makes a crucial assumption - *that both parties are symmetrically provisioned with massive computing resources.* For instance, Kreuter et al. [25] rely on the Ranger cluster at the Texas Advanced Computing Center to compute their results using 512 cores. In reality, the extent of a user's computing power may be their mobile phone, which has many orders of magnitude less computational ability. Moreover, even with access to a public compute cloud such as Amazon EC2 or Windows Azure, the sensitive nature of the user's data and the history of data leakage from cloud services [40, 42] prevent the direct porting of known SFE techniques.

In this paper, we develop mechanisms for the secure outsourcing of SFE computation from constrained devices to more capable infrastructure. Our protocol maintains the privacy of both participant's inputs and outputs while significantly reducing the computation and network overhead required by the mobile device for garbled circuit evaluation. We develop a number of extensions to allow the mobile device to check for malicious behavior from the circuit generator or the cloud and a novel Outsourced Oblivious Transfer for sending garbled input data to the cloud. We then implement the new protocol on a commodity mobile device and reasonably provisioned servers and demonstrate significant performance improvements over evaluating garbled circuits directly on the mobile device.

We make the following contributions:

- **Outsourced oblivious transfer & outsourced consistency checks:** Instead of blindly trusting the cloud with sensitive inputs, we develop a highly

efficient Outsourced Oblivious Transfer primitive that allows mobile devices to securely delegate the majority of computation associated with oblivious transfers. We also provide mechanisms to outsource consistency checks to prevent a malicious circuit generator from providing corrupt garbled values. These checks are designed in such a way that the computational load is almost exclusively on the cloud, but cannot be forged by a malicious or "lazy" cloud. We demonstrate that both of our additions are secure in the malicious model as defined by Kamara et al. [21].

- **Performance Analysis:** Extending upon the implementation by Kreuter et al. [25], we conduct an extensive performance analysis against a number of simple applications (e.g., edit distance) and cryptographic benchmarks (e.g., AES-128). Our results show that outsourcing SFE provides improvements to both execution time and bandwidth overhead. For the edit distance problem of size 128, we reduce execution time by 98.92% and bandwidth by 99.95% compared to direct execution without outsourcing on the mobile device.
- **Privacy Preserving Navigation App:** To demonstrate the practical need for our techniques, we design and implement an outsourced version of Dijkstra's shortest path algorithm as part of a Navigation mobile app. Our app provides directions for a Presidential motorcade without exposing its location, destination, or known hazards that should be avoided (but remain secret should the mobile device be compromised). *The optimized circuits generated for this app represent the largest circuits evaluated to date.* Without our outsourcing techniques, such an application is far too processor, memory and bandwidth intensive for any mobile phone.

While this work is similar in function and provides equivalent security guarantees to the Salus protocols recently developed by Kamara et al. [21], our approach is dramatically different. The Salus protocol framework builds their scheme on a completely different assumption, specifically, that they are outsourcing work from low-computation devices with *high communication bandwidth*. With provider-imposed bandwidth caps and relatively slow and unreliable cellular data connections, this is not a realistic assumption when developing solutions in the mobile environment. Moreover, rather than providing a proof-of-concept work demonstrating that offloading computation is possible, this work seeks to develop and thoroughly demonstrate the practical potential for evaluating large garbled circuits in a resource-constrained mobile environment.

The remainder of this work is organized as follows: Section 2 presents important related work and discusses how this paper differs from Salus; Section 3 provides cryptographic assumptions and definitions; Section 4 formally describes our protocols; Section 5 provides security discussion - we direct readers to our technical report [6] for full security proofs; Section 6 shows the results of our extensive performance analysis; Section 7 presents our privacy preserving navigation application for mobile phones; and Section 8 provides concluding remarks.

2 Related Work

Beginning with Fairplay [32], several secure two-party computation implementations and applications have been developed using Yao garbled circuits [43] in the semi-honest adversarial model [3, 15, 17, 19, 26, 28, 31, 38]. However, a malicious party using corrupted inputs or circuits can learn more information about the other party's inputs in these constructions [23]. To resolve these issues, new protocols have been developed to achieve security in the malicious model, using cut-and-choose constructions [30], input commitments [41], and other various techniques [22,34]. To improve the performance of these schemes in both the malicious and semi-honest adversarial models, a number of circuit optimization techniques have also been developed to reduce the cost of generating and evaluating circuits [8, 11, 24, 35]. Kreuter et al. [25] combined several of these techniques into a general garbled circuit protocol that is secure in the malicious model and can efficiently evaluate circuits on the order of billions of gates using parallelized server-class machines. This SFE protocol is currently the most efficient implementation that is fully secure in the malicious model. (The dual execution construction by Huang et al. leaks one bit of input [16].)

Garbled circuit protocols rely on oblivious transfer schemes to exchange certain private values. While several OT schemes of various efficiencies have been developed [1, 30, 36, 39], Ishai et al. demonstrated that any of these schemes can be extended to reduce k^c oblivious transfers to k oblivious transfers for any given constant c [18]. Using this extension, exchanging potentially large inputs to garbled circuits became much less costly in terms of cryptographic operations and network overhead. Even with this drastic improvement in efficiency, oblivious transfers still tend to be a costly step in evaluating garbled circuits.

Currently, the performance of garbled circuit protocols executed directly on mobile devices has been shown to be feasible only for small circuits in the semi-honest adversarial model [5, 13]. While outsourcing general computation to the cloud has been widely considered for improving the efficiency of applications running on mobile devices, the concept has yet to be widely applied to cryp-

tographic constructions. Green et al. began exploring this idea by outsourcing the costly decryption of ABE ciphertexts to server-class machines while still maintaining data privacy [12]. Considering the costs of exchanging inputs and evaluating garbled circuits securely, an outsourcing technique would be useful in allowing limited capability devices to execute SFE protocols. Naor et al. [37] develop an oblivious transfer technique that sends the chooser's private selections to a third party, termed a proxy. While this idea is applied to a limited application in their work, it could be leveraged more generally into existing garbled circuit protocols. Our work develops a novel extension to this technique to construct a garbled circuit evaluation protocol that securely outsources computation to the cloud.

In work performed concurrently and independently from our technique, Kamara et al. recently developed two protocols for outsourcing secure multiparty computation to the cloud in their Salus system [21]. While their work achieves similar functionality to ours, we distinguish our work in the following ways: first, their protocol is constructed with the assumption that they are outsourcing work from devices with low-computation but high-bandwidth capabilities. With cellular providers imposing bandwidth caps on customers and cellular data networks providing highly limited data transmission speed, we construct our protocol without this assumption using completely different cryptographic constructions. Second, their work focuses on demonstrating outsourced SFE as a proof-of-concept. Our work offers a rigorous performance analysis on mobile devices, and outlines a practical application that allows a mobile device to participate in the evaluation of garbled circuits that are orders of magnitude larger than those evaluated in the Salus system. Finally, their protocol that is secure in the malicious model requires that all parties share a secret key, which must be generated in a secure fashion before the protocol can be executed. Our protocol does not require any shared information prior to running the protocol, reducing the overhead of performing a multiparty fair coin tossing protocol a priori. While our work currently considers only the two-party model, by not requiring a preliminary multiparty fair coin toss, expanding our protocol to more parties will not incur the same expense as scaling such a protocol to a large number of participants. To properly compare security guarantees, we apply their security definitions in our analysis.

3 Assumptions and Definitions

To construct a secure scheme for outsourcing garbled circuit evaluation, some new assumptions must be considered in addition to the standard security measures taken in a two-party secure computation. In this section, we discuss the intuition and practicality of assuming a non-colluding cloud, and we outline our extensions on standard techniques for preventing malicious behavior when evaluating garbled circuits. Finally, we conclude the section with formal definitions of security.

3.1 Non-collusion with the cloud

Throughout our protocol, we assume that none of the parties involved will ever collude with the cloud. This requirement is based in theoretical bounds on the efficiency of garbled circuit evaluation and represents a realistic adversarial model. The fact that theoretical limitations exist when considering collusion in secure multiparty computation has been known and studied for many years [2, 7, 27], and other schemes considering secure computation with multiple parties require similar restrictions on who and how many parties may collude while preserving security [4, 9, 10, 20, 21]. Kamara et al. [21] observe that if an outsourcing protocol is secure when both the party generating the circuit and the cloud evaluating the circuit are malicious and colluding, this implies a secure two-party scheme where one party has sub-linear work with respect to the size of the circuit, which is currently only possible with fully homomorphic encryption. However, making the assumption that the cloud will not collude with the participating parties makes outsourcing securely a theoretical possibility. In reality, many cloud providers such as Amazon or Microsoft would not allow outside parties to control or affect computation within their cloud system for reasons of trust and to preserve a professional reputation. In spite of this assumption, we cannot assume the cloud will always be semi-honest. For example, our protocol requires a number of consistency checks to be performed by the cloud that ensure the participants are not behaving maliciously. Without mechanisms to force the cloud to make these checks, a "lazy" cloud provider could save resources by simply returning that all checks verified without actually performing them. Thus, our adversarial model encompasses a non-colluding but potentially malicious cloud provider that is hosting the outsourced computation.

3.2 Attacks in the malicious setting

When running garbled circuit based secure multiparty computation in the malicious model, a number of well-documented attacks exist. We address here how our system counters each.

Malicious circuit generation: In the original Yao garbled circuit construction, a malicious generator can garble a circuit to evaluate a function f' that is not the function f agreed upon by both parties and could compromise the security of the evaluator's input. To counter this, we

employ an extension of therandom seed techniquedeveloped by Goyal et al. [11] and implemented by Kreuter et al. [25]. Essentially, thetechniqueusesacut-and-choose, where the generator commits to a set of circuits that all presumably compute thesamefunction. Theparties then use a fair coin toss to select some of the circuits to be evaluated and some that will be re-generated and hashed by the cloud given the random seeds used to generate them initially. Theevaluating party then inspects the circuit commitments and compares them to the hash of the regenerated circuits to verify that all the check circuits were generated properly.

Selective failure attack: If, when the generator is send-ing the evaluator's garbled inputs during the oblivious transfer, heletstheevaluator choosebetween avalid garbled input bit and a corrupted garbled input, the evaluator'sability to complete the circuit evaluation will reveal to thegenerator which input bit wasused. To prevent this attack, weusetheinput encoding technique from Lindell and Pinkas [29], which lets the evaluator encode her input in such a way that a selective failure of the circuit reveals nothing about the actual input value. To prevent thegenerator from swapping garbled wirevalues, weuse acommitment techniqueemployed by Kreuter et al. [25].

Input consistency: Since multiple circuits are evaluated to ensure that a majority of circuits are correct, it is possible for either party to input different inputs to different evaluation circuits, which could reveal information about the other party's inputs. To keep the evaluator's inputs consistent, we again use the technique from Lindell and Pinkas [29], which sends all garbled inputs for every evaluation circuit in one oblivious transfer execution. To keep the generator's inputs consistent, we use the malleable claw-free collection construction of shelat and Shen [41]. This technique is described in further detail in Section 4.

Output consistency: When evaluating a two-output function, we ensure that outputs of both parties are kept private from the cloud using an extension of the technique developed by Kiraz [23]. The outputs of both parties are XORed with random strings within the garbled circuit, and the cloud uses a witness-indistinguishable zero-knowledge proof as in the implementation by Kreuter et al. [25]. This allows the cloud to choose a majority output valuewithout learning either party'soutput or undetectably tampering with the output. At the same time, the witness-indistinguishable proofs prevent either party from learning the index of the majority cir-cuit. This prevents the generator from learning anything by knowing which circuit evaluated to the majority out-put value.

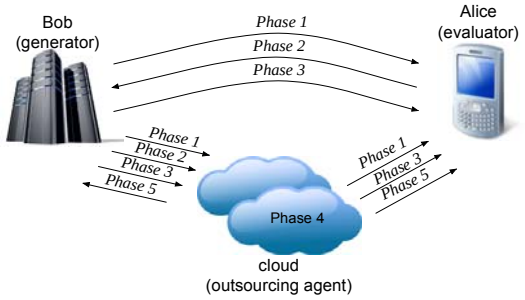


Figure 1: The complete outsourced SFE protocol.

3.3 Malleable claw-free collections

To prevent the generating party from providing differ-ent inputs for each evaluation circuit, we implement the malleable claw-free collections technique developed by shelat and Shen [41]. Their construction essentially allows the generating party to prove that all of the garbled input values were generated by exactly one function in a function pair, while the ability to find an element that is generated by both functions implies that the genera-tor can find a claw. It is composed of a four-tuple of algorithms (G,D,F,R), where G is the index selection algorithm for selecting a specific function pair, D is an algorithm for sampling from the domain of the function pair, F isthealgorithm for evaluating thefunctionsin the pair (in which it should bedifficult to find aclaw), and R is the "malleability" function. The function R maps elements from the domain of F to the range of F such that for $b2 \{0,1\}$,any I in the range of G,andany m_1,m_2 in the domain of F, we have for the function indexed by I and $bf_I^b(m_1?m_2)= f_I^b(m_1)\ \tilde{}\ R_I\ (m_2)$, where$?$ and $\tilde{}$ represent thegroup operations over thedomain and rangeof F. Weprovidefull definitions of their construction in our technical report [6].

3.4 Model and Definitions

The work of Kamara et al. [21] presents a definition of security based on the ideal-model/real-model secu-rity definitions common in secure multiparty computa-tion. Because their definition formalizes the idea of a non-colluding cloud, we apply their definitions to our protocol for the two-party case in particular. We summarize their definitions below. Real-model execution. The protocol takes place be-tween two parties (P_1,P_2) executing the protocol and a server P_3, where each of the executing parties provides input x_i, auxiliary input z_i, and random coins r_i and the server provides only auxiliary input z_3 and random coins r_3. In the execution, there exists somesubset of indepen-dent parties $(A_1,..,A_m),m \leq 3$ that are malicious adver-saries. Each adversary corrupts one executing party and

does not share information with other adversaries. For all honest parties, let OUT_i be its output, and for corrupted parties let OUT_i be its view of the protocol execution. The i^{th} partial output of a real execution is defined as:

$$REAL^{(i)}(k,x;r) = \{OUT_j : j \in H\} \cup OUT_i$$

where H is the set of honest parties and r is all random coins of all players.

Ideal-model execution. In the ideal model, the setup of participants is the same except that all parties are interacting with a trusted party that evaluates the function. All parties provide inputs x_i, auxiliary input z_i, and random coins r_i. If a party is semi-honest, it provides its actual inputs to the trusted party, while if the party is malicious (and non-colluding), it provides arbitrary input values. In the case of the server P_3, this means simply providing its auxiliary input and random coins, as no input is provided to the function being evaluated. Once the function is evaluated by the trusted third party, it returns the result to the parties P_1 and P_2, while the server P_3 does not receive the output. If a party aborts early or sends no input, the trusted party immediately aborts. For all honest parties, let OUT_i be its output to the trusted party, and for corrupted parties let OUT_i be some value output by P_i. The *ith* partial output of an ideal execution in the presence of some set of independent simulators is defined as:

$$IDEAL^{(i)}(k,x;r) = \{OUT_j : j \in H\} \cup OUT_i$$

where H is the set of honest parties and r is all random coins of all players. In this model, the formal definition of security is as follows:

Definition 1. *A protocol securely computes a function f if there exists a set of probabilistic polynomial-time (PPT) simulators $\{Sim_i\}_{i\in[3]}$ such that for all PPT adversaries $(A_1,...,A_3)$, x, z, and for all $i \in [3]$:*

$$\{REAL^{(i)}(k,x;r)\}_{k\in N} \overset{c}{\approx} \{IDEAL^{(i)}(k,x;r)\}_{k\in N}$$

Where $S = (S_1,...,S_3)$, $S_i = Sim_i(A_i)$, and r is random and uniform.

4 Protocol

Our protocol can be divided into five phases, illustrated in Figure 1. Given a circuit generator Bob, and an evaluating mobile device Alice, the protocol can be summarized as follows:

- Phase 1: Bob generates a number of garbled circuits, some of which will be checked, others will be evaluated. After Bob commits to the circuits, Alice and Bob use a fair coin toss protocol to select which circuits will be checked or evaluated. For the check circuits, Bob sends the random seeds used to generate the circuits to the cloud and the hashes of each circuit to Alice. These are checked to ensure that Bob has not constructed a circuit that is corrupted or deviates from the agreed-upon function.
- Phase 2: Alice sends her inputs to Bob via an outsourced oblivious transfer. Bob then sends the corresponding garbled inputs to the cloud. This allows the cloud to receive Alice's garbled inputs without Bob or the cloud ever learning her true inputs.
- Phase 3: Bob sends his garbled inputs to the cloud, which verifies that they are consistent for each evaluation circuit. This prevents Bob from providing different inputs to different evaluation circuits.
- Phase 4: The cloud evaluates the circuit given Alice and Bob's garbled inputs. Since the cloud only sees garbled values during the evaluation of the circuit, it never learns anything about either party's input or output. Since both output values are blinded with one-time pads, they remain private even when the cloud takes a majority vote.
- Phase 5: The cloud sends the encrypted output values to Alice and Bob, who are guaranteed its authenticity through the use of commitments and zero-knowledge proofs.

Inputs: Alice has a string of encoded input bits ea of length $\ell \cdot n$ and Bob has pairs of input values $(x_{0,j}, x_{1,j})$ for $j = 1...\ell \cdot n$.

1. **Setup:** Alice generates random matrix T of size $\ell \cdot n \times t$, Bob generates random string s of length t.
2. **Primitive OT:** Alice and Bob execute t 1-out-of-2 oblivious transfers with Alice inputting $(T^i, T^i \oplus ea)$ and Bob inputting selection bits s (T^i denotes the i^{th} column of the T matrix). Bob sets the resulting columns as matrix Q.
3. **Permuting the output:** Alice generates random string p of length $\ell \cdot n$ and sends it to Bob.
4. **Encrypting the output:** Bob sets the encrypted output pairs $y_{0,j}, y_{1,j}$ where $y_{b,j} = x_{b,j} \oplus H_1(j, Q_j \oplus (b \cdot s))$ (Q_j denotes the j^{th} row of the Q matrix).
5. **Permuting the outputs:** Bob permutes the encrypted output pairs as $y_{0\oplus p_j,j}, y_{1\oplus p_j,j}$ and sends the resulting set of pairs Y to the cloud.
6. **Decrypting the output:** Alice sends $h = ea \oplus p$ and T to the cloud. The cloud recovers $z_j = y_{h_j,j} \oplus H_1(j, T_j)$ for $j = 1...\ell \cdot n$ (T_j denotes the j^{th} row of the T matrix).

Figure 2: The Outsourced Oblivious Transfer protocol

4.1 Participants

Our protocols reference three different entities:
Evaluator: The evaluating party, called Alice, is assumed to be a mobile device that is participating in a secure two-party computation.
Generator: The party generating the garbled circuit, called Bob, is an application- or web- server that is the second party participating with Alice in the secure computation.
Proxy: The proxy, called cloud, is a third party that is performing heavy computation on behalf of Alice, but is not trusted to know her input or the function output.

4.2 Outsourced Protocol

Common inputs: a function $f(x,y)$ that is to be securely computed, a claw-free collection $(G_{CLW}, D_{CLW}, F_{CLW}, R_{CLW})$, two hash functions $H_1 : \{0,1\}^* \rightarrow \{0,1\}^n$ and $H_2 : \{0,1\}^* \rightarrow \{0,1\}^w$, a primitive 1-out-of-2 oblivious transfer protocol, a perfectly hiding commitment scheme $com_H(key, message)$, and security parameters for the number of circuits built k, the number of primitive oblivious transfers t, and the number of encoding bits for each of Alice's input wires ℓ.
Private inputs: The generating party Bob inputs a bit string b and a random string of bits b_r that is the length of the output string. The evaluating party Alice inputs a bit string a and a random string of bits a_r that is the length of the output string. Assume without loss of generality that all input and output strings are of length $|a| = n$.
Output: The protocol outputs separate private values fa for Alice and fb for Bob.
Phase 1: Circuit generation and checking

1. *Circuit preparation:* Before beginning the protocol, both parties agree upon a circuit representation of the function $f(a,b)$, where the outputs of the function may be defined separately for Alice and Bob as $f_A(a,b)$ and $f_B(a,b)$. The circuit must also meet the following requirements:
   (a) Additional XOR gates must be added such that Bob's output is set to $fb = f_B(a,b) \oplus b_r$ and Alice's output is set to $fa = f_A(a,b) \oplus a_r$.
   (b) For each of Alice's input bits, the input wire w_i is split into ℓ different input wires $w_{j,i}$ such that $w_i = w_{1,i} \oplus w_{2,i} \oplus ... \oplus w_{l,i}$ following the input encoding scheme by Lindell and Pinkas [29]. This prevents Bob from correlating a selective failure attack with any of Alice's input bit values.

2. *Circuit garbling:* the generating party, Bob, constructs k garbled circuits using a circuit garbling technique $Garble(\cdot,\cdot)$. When given a circuit representation C of a function and random coins rc, $Garble(C,rc)$ outputs a garbled circuit GC that evaluates C. Given the circuit C and random coins $rc_1...rc_k$, Bob generates garbled circuits $Garble(C,rc_i) = GC_i$ for $i = 1...k$. For Bob's j^{th} input wire on the i^{th} circuit, Bob associates the value $H_2(\beta_{b,j,i})$ with the input value b, where $\beta_{b,j,i} = F_{CLW}(b, I, \alpha_{b,j,i})$. For Alice's j^{th} input wire, Bob associates the value $H_2(\delta_{b,j,i})$ with the input value b, where $\delta_{b,j,i} = F_{CLW}(b, I, \gamma_{b,j,i})$. All the values $\alpha_{b,j,i}$ and $\gamma_{b,j,i}$ for $b = \{0,1\}, j = 1...n, i = 1...k$ are selected randomly from the domain of the claw-free pair using D.

3. *Circuit commitment:* Bob generates commitments for all circuits by hashing $H_1(GC_i) = HC_i$ for $i = 1...k$. Bob sends these hashes to Alice. In addition, for every output wire $w_{b,j,i}$ for $b = \{0,1\}$, $j = 1...n$ and $i = 1...k$, Bob generates commitments $CO_{j,i} = com_H(ck_{j,i}, (H_2(w_{0,j,i}), H_2(w_{1,j,i})))$ using commitment keys $ck_{j,i}$ for $j = 1...n$ and $i = 1...k$ and sends them to both Alice and the cloud.

4. *Input label commitment:* Bob commits to Alice's garbled input values as follows: for each generated circuit $i = 1...k$ and each of Alice's input wires $j = 1...\ell \cdot n$, Bob creates a pair of commitment keys $ik_{0,j,i}, ik_{1,j,i}$ and commits to the input wire label seeds $\delta_{0,j,i}$ and $\delta_{1,j,i}$ as $CI_{b,j,i} = com_H(ik_{b,j,i}, \delta_{b,j,i})$. For each of Alice's input wires $j = 1...\ell \cdot n$, Bob randomly permutes the commitments within the pair $CI_{0,j,i}, CI_{1,j,i}$ across every $i = 1...k$. This prevents the cloud from correlating the location of the commitment with Alice's input value during the OOT phase.

5. *Cut and choose:* Alice and Bob then run a fair coin toss protocol to agree on a set of circuits that will be evaluated, while the remaining circuits will be checked. The coin toss generates a set of indices $Chk \subset \{1,...,k\}$ such that $|Chk| = \frac{3}{5}k$, as in shelat and Shen's cut-and-choose protocol [41]. The remaining indices are placed in the set Evl for evaluation, where $|Evl| = e = \frac{2}{5}k$. For every $i \in Chk$, Bob sends rc_i and the values $[\alpha_{b,1,i}, ..., \alpha_{b,n,i}]$ and $[\gamma_{b,1,i}, ..., \gamma_{b,\ell \cdot n,i}]$ for $b = \{0,1\}$ to the cloud. Bob also sends all commitment keys $ck_{j,i}$ for $j = 1...n$ and $i \in Chk$ to the cloud. Finally, Bob sends the commitment keys $ik_{b,j,i}$ for $b = \{0,1\}$, $i \in Chk$, and $j = 1...\ell \cdot n$ to the cloud. The cloud then generates $Garble(C,rc_i) = GC'_i$ for $i \in Chk$. For each $i \in Chk$, the cloud then hashes each check circuit $H_1(GC'_i) = HC'_i$ and checks that:

- each commitment $CO_{j,i}$ for $j = 1...n$ is well formed
- the value $H_2(\beta_{b,j,i})$ is associated with the input value b for Bob's j^{th} input wire
- the value $H_2(\delta_{b,j,i})$ is associated with the input value b for Alice's j^{th} input wire
- for every bit value b and input wire j, the values committed in $CI_{b,j,i}$ are correct

If any of these checks fail, the cloud immediately aborts. Otherwise, it sends the hash values HC'_i for $i \in Chk$ to Alice. For every $i \in Chk$, Alice checks if $HC_i = HC'_i$. If any of the hash comparisons fail, Alice aborts.

Phase 2: Outsourced Oblivious Transfer (OOT)

1. *Input encoding:* For every bit $j = 1...n$ in her input a, Alice sets encoded input ea_j as a random string of length ℓ such that $ea_{1,j} \oplus ea_{2,j} \oplus ... \oplus ea_{l,j} = a_j$ for each bit in ea_j. This new encoded input string ea is of length $\ell \cdot n$.

2. *OT setup:* Alice initializes an $\ell \cdot n \times t$ matrix T with uniformly random bit values, while Bob initializes a random bit vector s of length t. See Figure 2 for a more concise view.

3. *Primitive OT operations:* With Alice as the sender and Bob as the chooser, the parties initiate t 1-out-of-2 oblivious transfers. Alice's input to the i^{th} instance of the OT is the pair $(T^i, T^i \oplus ea)$ where T^i is the i^{th} column of T, while Bob's input is the i^{th} selection bit from the vector s. Bob organizes the t selected columns as a new matrix Q.

4. *Permuting the selections:* Alice generates a random bit string p of length $\ell \cdot n$, which she sends to Bob.

5. *Encrypting the commitment keys:* Bob generates a matrix of keys that will open the committed garbled input values and proofs of consistency as follows: for Alice's j^{th} input bit, Bob creates a pair $(x_{0,j}, x_{1,j})$, where $x_{b,j} = [ik_{b,j,Evl_1}, ik_{b,j,Evl_2}, ..., ik_{b,j,Evl_e}]||[\gamma_{b_j,j,Evl_2} \star (\gamma_{b_j,j,Evl_1})^{-1}, \gamma_{b_j,j,Evl_3} \star (\gamma_{b_j,j,Evl_1})^{-1}, ..., \gamma_{b_j,j,Evl_e} \star (\gamma_{b_j,j,Evl_1})^{-1}]$ and Evl_i denotes the i^{th} index in the set of evaluation circuits. For $j = 1...\ell \cdot n$, Bob prepares $(y_{0,j}, y_{1,j})$ where $y_{b,j} = x_{b,j} \oplus H_1(j, Q_j \oplus (b \cdot s))$. Here, Q_j denotes the j^{th} row in the Q matrix. Bob permutes the entries using Alice's permutation vector as $(y_{0 \oplus p_j,j}, y_{1 \oplus p_j,j})$. Bob sends this permuted set of ciphertexts Y to the cloud.

6. *Receiving Alice's garbled inputs:* Alice blinds her input as $h = ea \oplus p$ and sends h and T to the cloud. The cloud recovers the commitment keys and consistency proofs $x_{b,j} = y_{h_j,j} \oplus H_1(j, T_j)$ for $j = 1...\ell \cdot n$. Here, h_j denotes the j^{th} bit of the string h and T_j denotes the j^{th} row in the T matrix. Since for every $j \in Evl$, the cloud only has the commitment key for the b garbled value (not the $b \oplus 1$ garbled value), the cloud can correctly decommit only the garbled labels corresponding to Alice's input bits.

7. *Verifying consistency across Alice's inputs:* Given the decommitted values $[\delta_{b,1,i}, ..., \delta_{b,\ell \cdot n,i}]$ and the modified pre images $[\gamma_{b_j,j,Evl_2} \star (\gamma_{b_j,j,Evl_1})^{-1}, \gamma_{b_j,j,Evl_3} \star (\gamma_{b_j,j,Evl_1})^{-1}, ..., \gamma_{b_j,j,Evl_e} \star (\gamma_{b_j,j,Evl_1})^{-1}]$, the cloud checks that:

$$\delta_{b_j,j,i} = \delta_{b_j,j,Evl_1} \diamond R_{CLW}(I, \gamma_{b_j,j,i} \star (\gamma_{b_j,j,Evl_1})^{-1})$$

for $i = 2...e$. If any of these checks fails, the cloud aborts the protocol.

Phase 3: Generator input consistency check

1. *Delivering inputs:* Bob delivers the hash seeds for each of his garbled input values $[\beta_{b_1,1,i}, \beta_{b_2,2,i}, ..., \beta_{b_n,n,i}]$ for every evaluation circuit $i \in Evl$ to the cloud, which forwards a copy of these values to Alice. Bob then proves the consistency of his inputs by sending the modified preimages $[\alpha_{b_j,j,Evl_2} \star (\alpha_{b_j,j,Evl_1})^{-1}, \alpha_{b_j,j,Evl_3} \star (\alpha_{b_j,j,Evl_1})^{-1}, ..., \alpha_{b_j,j,Evl_e} \star (\alpha_{b_j,j,Evl_1})^{-1}]$ such that $F_{CLW}(b_i, I, \alpha_{b_i,j,i}) = \beta_{b_i,j,i}$ for $j = 1...n$ and $i \in Evl$ such that GC_i was generated with the claw-free function pair indexed at I.

2. *Check consistency:* Alice then checks that all the hash seeds were generated by the same function by checking if:

$$\beta_{b_j,j,i} = \beta_{b_j,j,Evl_1} \diamond R_{CLW}(I, \alpha_{b_j,j,i} \star (\alpha_{b_j,j,Evl_1})^{-1})$$

for $i = 2...e$. If any of these checks fails, Alice aborts the protocol.

Phase 4: Circuit evaluation

1. *Evaluating the circuit:*For each evaluation circuit, the cloud evaluates $GC_i(ga_i, gb_i)$ for $i \in Evl$ in the pipelined manner described by Kreuter et al. in [25]. Each circuit produces two garbled output strings, (gfa_i, gfb_i).

2. *Checking the evaluation circuits:* Once these output have been computed, the cloud hashes each evaluation circuit as $H_1(GC_i) = HC'_i$ for $i \in Evl$ and sends these hash values to Alice. Alice checks that for every $i, HC_i = HC'_i$. If any of these checks do not pass, Alice aborts the protocol.

Phase 5: Output check and delivery

1. *Committing the outputs:*The cloud then generates random commitment keys ka_i, kb_i and commits the output values to their respective parties according to the commitment scheme defined by Kiraz [23], generating $CA_{j,i} = commit(ka_{j,i}, gfa_{j,i})$ and $CB_{j,i} = commit(kb_{j,i}, gfb_{j,i})$ for $j = 1...n$ and $i = 1...e$. The cloud then sends all *CA* to Alice and *CB* to Bob.

2. *Selection of majority output:* Bob opens the commitments $CO_{j,i}$ for $j = 1...n$ and $i = 1...e$ for both Alice and the Cloud. These commitments contain the mappings from the hash of each garbled output wire $H_2(w_{b,j,i})$ to real output values $b_{j,i}$ for $j = 1...n$ and $i = 1...e$. The cloud selects a circuit index maj such that the output of that circuit matches the majority of outputs for both Alice and Bob. That is, $fa_{maj} = fa_i$ and $fb_{maj} = fb_i$ for i in a set of indices *IND* that is of size $|IND| > \frac{e}{2}$

3. *Proof of output consistency:* Using the OR-proofs as described by Kiraz [23], the cloud proves to Bob that *CB* contains valid garbled output bit values based on the de-committed output values from the previous step. The cloud then performs the same proof to Alice for her committed values *CA*. Note that these proofs guarantee the output was generated by one of the circuits, but the value maj remains hidden from both Alice and Bob.

4. *Output release:* The cloud then decommits gfa_{maj} to Alice and gfb_{maj} to Bob. Given these garbled outputs and the bit values corresponding to the hash of each output wire, Alice recovers her output string fa, and Bob recovers his output string fb.

5. *Output decryption:* Alice recovers her output $f_A(a,b) = fa \oplus a_r$, while Bob recovers $f_B(a,b) = fb \oplus b_r$.

5 Security Guarantees

In this section, we provide a summary of the security mechanisms used in our protocol and an informal security discussion of our new outsourced oblivious transfer primitive. Due to space limitations, we provide further discussion and proofs of security in our technical report [6].

Recall from Section 3 that there are generally four security concerns when evaluating garbled circuits in the malicious setting. To solve the problem of malicious circuit generation, we apply the random seed check variety of cut-&-choose developed by Goyal et al. [11]. To solve the problem of selective failure attacks, we employ the input encoding technique developed by Lindell and Pinkas [29]. To prevent an adversary from using inconsistent inputs across evaluation circuits, we employ the witness-indistinguishable proofs from shelat and Shen [41]. Finally, to ensure the majority output value is selected and not tampered with, we use the XOR-and-prove technique from Kiraz [23] as implemented by Kreuter et al. [25]. In combination with the standard semi-honest security guarantees of Yao garbled circuits, these security extensions secure our scheme in the malicious security model.

Outsourced Oblivious Transfer: Our outsourced oblivious transfer is an extension of a technique developed by Naor et al. [37] that allows the chooser to select entries that are forwarded to a third party rather than returned to the chooser. By combining their concept of a proxy oblivious transfer with the semi-honest OT extension by Ishai et al. [18], our outsourced oblivious transfer provides a secure OT in the malicious model. We achieve this result for four reasons:

1. First, since Alice never sees the outputs of the OT protocol, she cannot learn anything about the garbled values held by the generator. This saves us from having to implement Ishai's extension to prevent the chooser from behaving maliciously.
2. Since the cloud sees only random garbled values and Alice's input blinded by a one-time pad, the cloud learns nothing about Alice's true inputs.
3. Since Bob's view of the protocol is almost identical to his view in Ishai's standard extension, the same security guarantees hold (i.e., security against a malicious sender).
4. Finally, if Alice does behave maliciously and uses inconsistent inputs to the primitive OT phase, there is a negligible probability that those values will hash to the correct one-time pad keys for recovering *either* commitment key, which will prevent the cloud from de-committing the garbled input values.

It is important to note that this particular application of the OOT allows for this efficiency gain since the evaluation of the garbled circuit will fail if Alice behaves maliciously. By applying the maliciously secure extension by Ishai et al. [18], this primitive could be applied generally as an oblivious transfer primitive that is secure in the malicious model. Further discussion and analysis of this general application is outside the scope of this work.

We provide the following security theorem here, which gives security guarantees identical to the Salus protocol by Kamara et al. [21]. However, we use different constructions and require a completely different proof, which is available in our technical report [6].

Theorem 1. *The outsourced two-party SFE protocol securely computes a function $f(a,b)$ in the following two*

corruption scenarios: (1)The cloud is malicious and non-cooperative with respect to the rest of the parties, while all other parties are semi-honest, (2)All but one party is malicious, while the cloud is semi-honest.

6 Performance Analysis

We now characterize how garbled circuits perform in the constrained-mobile environment with and without outsourcing.[1] Two of the most important constraints for mobile devices are computation and bandwidth, and we show that order of magnitude improvements for both factors are possible with outsourced evaluation. We begin by describing our implementation framework and testbed before discussing results in detail.

6.1 Framework and Testbed

Our framework is based on the system designed by Kreuter et al. [25], hereafter referred to as *KSS* for brevity. We implemented the outsourced protocol and performed modifications to allow for the use of the mobile device in the computation. Notably, *KSS* uses MPI [33] for communication between the multiple nodes of the multi-core machines relied on for circuit evaluation. Our solution replaces MPI calls on the mobile device with sockets that communicate directly with the Generator and Proxy. To provide a consistent comparison, we revised the *KSS* codebase to allow for direct evaluation between the mobile device (the Evaluator) and the cloud-based Generator.[2]

Our deployment platform consists of two Dell R610 servers, each containing dual 6-core Xeon processors with 32 GB of RAM and 300 GB 10K RPM hard drives, running the Linux 3.4 kernel and connected as a VLAN on an internal 1 Gbps switch. These machines perform the roles of the Generator and Proxy, respectively, as described in Section 4.1. The mobile device acts as the Evaluator. We use a Samsung Galaxy Nexus phone with a 1.2 GHz dual-core ARM Cortex-A9 processor and 1 GB of RAM, running the Android 4.0 "Ice Cream Sandwich" operating system. We connect an Apple Airport Express wireless access point to the switch attaching the servers, The Galaxy Nexus communicates to the Airport Express over an 802.11n 54Mbps WiFi connection in an isolated environment to minimize co-channel interference. All tests are run 10 times with error bars on figures representing 95% confidence intervals.

[1] We contacted the authors of the Salus protocol [21] in an attempt to acquire their framework to compare the performance of their scheme with ours, but they were unable to release their code.

[2] The full technical report [6] describes a comprehensive list of modifications and practical improvements made to *KSS*, including fixes that were added back into the codebase of *KSS* by the authors. We thank those authors for their assistance.

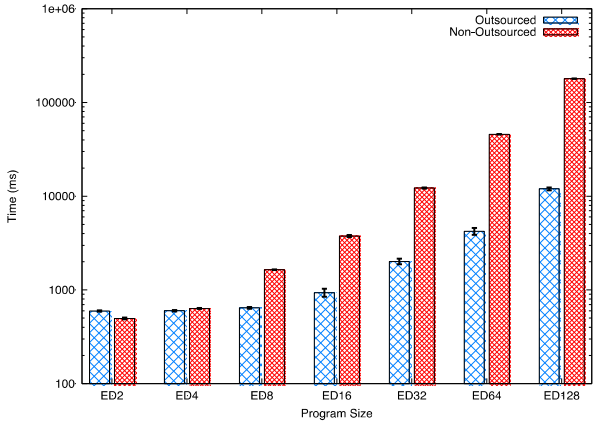


Figure 3: Execution time for the Edit Distance program of varying input sizes, with 2 circuits evaluated.

We measured both the total execution time of the programs and microbenchmarks for each program. All results are from the phone's standpoint. We do not measure the time the programs take to compile as we used the standard compiler from Kreuter et al. For our microbenchmarks, the circuit garbling and evaluation pair is referred to as the 'evaluation'.

6.2 Execution Time

Our tests evaluated the following problems:

Millionaires: This problem models the comparison of two parties comparing their net worth to determine who has more money without disclosing the actual values. We perform the test on input values ranging in size from 4 to 8192 bits.

Edit (Levenshtein) Distance: This is a string comparison algorithm that compares the number of modifications required to covert one string into another. We performed the comparison based on the circuit generated by Jha et al. [19] for strings sized between 4 and 128 bytes.

Set Intersection: This problem matches elements between the private sets of two parties without learning anything beyond the intersecting elements. We base our implementation on the SCS-WN protocol proposed by Huang et al. [14], and evaluate for sets of size 2 to 128.

AES: We compute AES with a 128-bit key length, based on a circuit evaluated by Kreuter et al. [25].

Figure 3 shows the result of the edit distance computation for input sizes of 2 to 128 with two circuits evaluated. This comparison represents worst-case operation due to the cost of setup for a small number of small circuits—with input size 2, the circuit is only 122 gates in size. For larger input sizes, however, outsourced computation becomes significantly faster. Note that the graph is logarithmic such that by the time strings of size 32 are evaluated, the outsourced execution is over 6 times

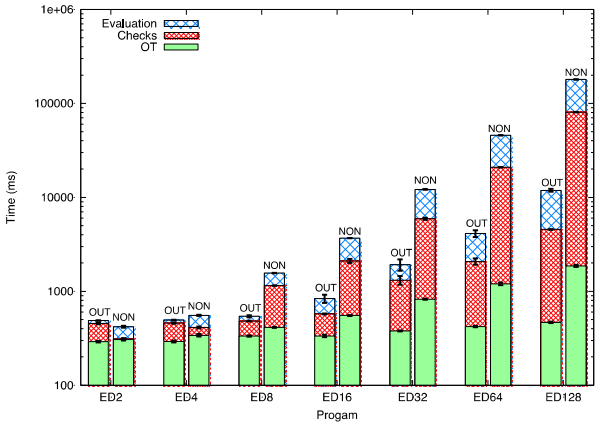


Figure 4: Execution time for significant stages of garbled circuit computation for outsourced and non-outsourced evaluation. The Edit Distance program is evaluated with variable input sizes for the two-circuit case.

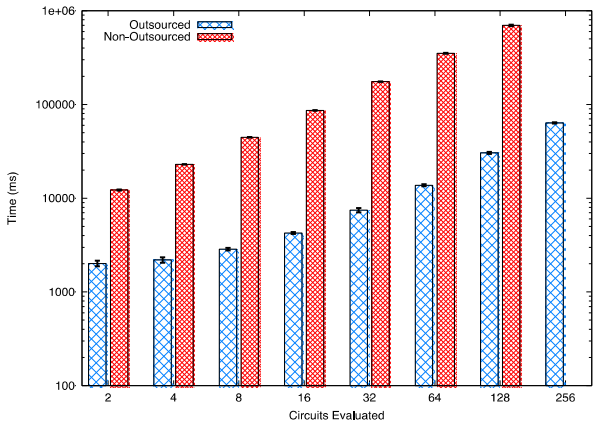


Figure 5: Execution time for the Edit Distance problem of size 32, with between 2 and 256 circuits evaluated. In the non-outsourced evaluation scheme, the mobile phone runs out of memory evaluating 256 circuits.

faster than non-outsourced execution, while for strings of size 128 (comprising over 3.4 million gates), outsourced computation is over 16 times faster.

The reason for this becomes apparent when we examine Figure 4. There are three primary operations that occur during the SFE transaction: the oblivious transfer (OT) of participant inputs, the circuit commit (including the circuit consistency check), and the circuit generation and evaluation pair. As shown in the figure, the OT phase takes 292 ms for input size 2, but takes 467 ms for input size 128. By contrast, in the non-outsourced execution, the OT phase takes 307 ms for input size 2, but increases to 1860 ms for input size 128. The overwhelming factor, however, is the circuit evaluation phase. It increases from 34 ms (input size 2) to 7320 ms (input size 128) for the outsourced evaluation, a 215 factor increase. For non-outsourced execution however, this phase increases from 108 ms (input size 2) to 98800 ms (input size 128), a factor of 914 increase.

6.3 Evaluating Multiple Circuits

The security parameter for the garbled circuit check is $2^{-0.32k}$ [25], where k is the number of generated circuits. To ensure a sufficiently low probability (2^{-80}) of evaluating a corrupt circuit, 256 circuits must be evaluated. However, there are increasing execution costs as increasing numbers of circuits are generated. Figure 5 shows the execution time of the Edit Distance problem of size 32 with between 2 and 256 circuits being evaluated. In the outsourced scheme, costs rise as the number of circuits evaluated increases. Linear regression analysis shows we can model execution time T as a function of the number of evaluated circuits k with the equation $T = 243.2k + 334.6$ ms, with a coefficient of determination R^2 of 0.9971. However, note that in the non-outsourced scheme, execution time increases over 10 times as quickly compared to outsourced evaluation. Regression analysis shows execution time $T = 5435.7k + 961$ ms, with $R^2 = 0.9998$. Because in this latter case, the mobile device needs to perform all computation locally as well as transmit all circuit data to the remote parties, these costs increase rapidly. Figure 6 provides more detail about each phase of execution. Note that the OT costs are similar between outsourced and non-outsourced execution for this circuit size, but that the costs of consistency checks and evaluation vastly increase execution time for non-outsourced execution.

Note as well that in the non-outsourced scheme, there are no reported values for 256 circuits, as the Galaxy Nexus phone ran out of memory before the execution completed. We observe that a single process on the phone is capable of allocating 512 MB of RAM before the phone would report an out of memory error, providing insight into how much intermediate state is required for non-outsourced evaluation. Thus, to handle circuits of any meaningful size with enough check circuits for a strong security parameter, the *only way* to be able to perform these operations is through outsourcing.

Table 1 presents the execution time of a representative subset of circuits that we evaluated. It spans circuits from small to large input size, and from 8 circuits evaluated to the 256 circuits required for a 2^{-80} security parameter. Note that in many cases it is impossible to evaluate the non-outsourced computation because of the mobile device's inability to store sufficient amounts of state. Note as well that particularly with complex circuits such as set intersection, even when the non-outsourced evaluation is capable of returning an answer, it can require orders of

	8 Circuits		32 Circuits		128 Circuits		256 Circuits	
Program	Outsourced	KSS	Outsourced	KSS	Outsourced	KSS	Outsourced	KSS
Millionaires 128	2150.0 ± 1%	6130.0 ± 0.6%	8210.0 ± 3%	23080.0 ± 0.6%	38100.0 ± 7%	91020.0 ± 0.8%	75700.0 ± 1%	180800.0 ± 0.5%
Millionaires 1024	4670.0 ± 6%	46290.0 ± 0.4%	17800.0 ± 1%	180500.0 ± 0.3%	75290.0 ± 1%	744500.0 ± 0.7%	151000.0 ± 1%	1507000.0 ± 0.5%
Millionaires 8192	17280.0 ± 0.9%	368800.0 ± 0.4%	76980.0 ± 0.5%	1519000.0 ± 0.4%	351300.0 ± 0.7%	-	880000.0 ± 20%	-
Edit Distance 2	1268.0 ± 0.9%	794.0 ± 1%	4060.0 ± 1%	2125.0 ± 0.7%	19200.0 ± 2%	7476.0 ± 0.5%	42840.0 ± 0.4%	14600.0 ± 0.8%
Edit Distance 32	2860.0 ± 3%	44610.0 ± 0.7%	7470.0 ± 5%	175600.0 ± 0.5%	30500.0 ± 3%	699000.0 ± 2%	63600.0 ± 1%	-
Edit Distance 128	12800.0 ± 2%	702400.0 ± 0.5%	30300.0 ± 2%	2805000.0 ± 0.8%	106200.0 ± 0.6%	-	213400.0 ± 0.3%	-
Set Intersection 2	1598.0 ± 0.8%	1856.0 ± 0.9%	5720.0 ± 0.7%	6335.0 ± 0.4%	26100.0 ± 2%	24420.0 ± 0.6%	56350.0 ± 0.8%	48330.0 ± 0.6%
Set Intersection 32	5200.0 ± 10%	96560.0 ± 0.6%	13800.0 ± 1%	400800.0 ± 0.6%	59400.0 ± 1%	-	125300.0 ± 0.9%	-
Set Intersection 128	24300.0 ± 2%	1398000.0 ± 0.4%	55400.0 ± 3%	5712000.0 ± 0.4%	1998000.0 ± 0.5%	-	395200.0 ± 0.8%	-
AES-128	2450.0 ± 2%	15040.0 ± 0.7%	9090.0 ± 5%	58920.0 ± 0.5%	39000.0 ± 2%	276200.0 ± 0.6%	81900.0 ± 1%	577900.0 ± 0.5%

Table 1: Execution time (in ms) of outsourced vs non-outsourced (KSS) evaluation for a subset of circuits. Results with a dash indicate evaluation that the phone was incapable of performing.

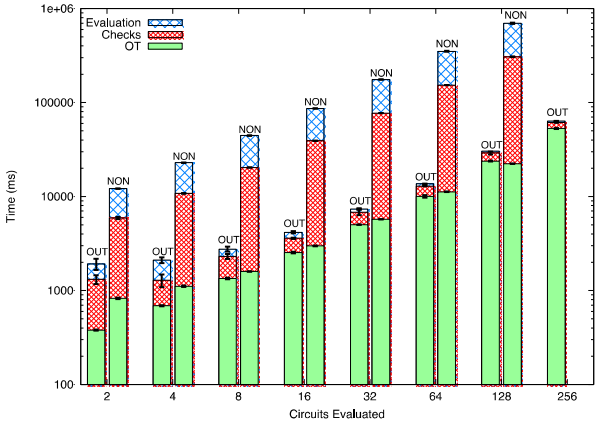


Figure 6: Microbenchmarks of execution time for Edit Distance with input size 32, evaluating from 2 to 256 circuits. Note that the y-axis is log-scale; consequently, the vast majority of execution time is in the check and evaluation phases for non-outsourced evaluation.

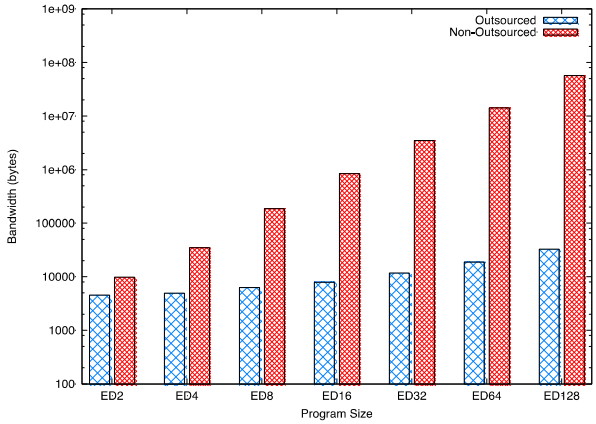


Figure 7: Bandwidth measurements from the phone to remote parties for the Edit Distance problem with varying input sizes, executing two circuits.

magnitude more time than with outsourced evaluation. For example, evaluating the set intersection problem with 128 inputs over 32 circuits requires just over 55 seconds for outsourced evaluation but *over an hour and a half* with the non-outsourced *KSS* execution scheme. Outsourced evaluation represents a time savings of 98.92%. For space concerns, we have omitted certain values; full results can be found in our technical report [6].

Multicore Circuit Evaluation We briefly note the effects of multicore servers for circuit evaluation. The servers in our evaluation each contain dual 6-core CPUs, providing 12 total cores of computation. The computation process is largely CPU-bound: while circuits on the servers are being evaluated, each core was reporting approximately 100% utilization. This is evidenced by regression analysis when evaluating between 2 and 12 circuit copies; we find that execution time $T = 162.6k + 1614.6$ ms, where k is the number of circuits evaluated, with a coefficient of determination R^2 of 0.9903. As the number of circuits to be evaluated increases beyond the number of available cores, the incremental costs of adding new circuits becomes higher; in our observation of execution time for 12 to 256 circuits, our regression analysis provided the equation $T = 247.4k - 410.6$ ms, with $R^2 = 0.998$. This demonstrates that evaluation of large numbers of circuits is optimal when every evaluated circuit can be provided with a dedicated core.

The results above show that as many-way servers are deployed in the cloud, it becomes easier to provide optimal efficiency computing outsourced circuits. A 256-core machine would be able to evaluate 256 circuits in parallel to provide the accepted standard 2^{-80} security parameter. Depending on the computation performed, there can be a trade-off between a slightly weaker security parameter and maintaining *optimal* evaluation on servers with lower degrees of parallelism. In our testbed, optimal evaluation with 12 cores provides a security parameter of $2^{-3.84}$. Clearly more cores would provide stronger security while keeping execution times proportional to our results. A reasonable trade-off might be 32 circuits, as 32-core servers are readily available. Evaluating 32 circuits provides a security parameter of $2^{-10.2}$, equivalent to the adversary having less than a $\frac{1}{512}$ chance of causing the evaluator to compute over a majority of corrupt circuits. Stronger security guarantees on less par-

	32 Circuits		Factor
Program	Outsourced	KSS	Improvement
Millionaires 128	336749	1445369	4.29X
Millionaires 1024	2280333	11492665	5.04X
Millionaires 8192	17794637	91871033	5.16X
Edit Distance 2	56165	117245	2.09X
Edit Distance 32	134257	41889641	312.01X
Edit Distance 128	350721	682955633	1947.29X
Set Intersection 2	117798	519670	4.41X
Set Intersection 32	1173844	84841300	72.28X
Set Intersection 128	4490932	1316437588	293.13X
AES-128	367364	9964576	27.12X

Table 2: Total Bandwidth (Bytes) transmitted to and from the phone during execution.

allel machines can be achieved at the cost of increasing execution time, as individual cores will not be dedicated to circuit evaluation. However, if a 256-core system is available, it will provide optimal results for achieving a 2^{-80} security parameter.

6.4 Bandwidth

For a mobile device, the costs of transmitting data are intrinsically linked to power consumption, as excess data transmission and reception reduces battery life. Bandwidth is thus a critical resource constraint. In addition, because of potentially uncertain communication channels, transmitting an excess of information can be a rate-limiting factor for circuit evaluation. Figure 7 shows the bandwidth measurement between the phone and remote parties for the edit distance problem with 2 circuits. When we compared execution time for this problem in Figure 3, we found that trivially small circuits could execute in less time without outsourcing. Note, however, that *there are no cases where the non-outsourced scheme consumes less bandwidth than with outsourcing*.

This is a result of the significant improvements garnered by using our outsourced oblivious transfer (OOT) construction described in Section 4. Recall that with the OOT protocol, the mobile device sends inputs for evaluation to the generator; however, after this occurs, the majority of computation until the final output verification from the cloud occurs between the generator and the cloud, with the mobile device only performing minor consistency checks. Figure 7 shows that the amount of data transferred increases only nominally compared to the non-outsourced protocol. Apart from the initial set of inputs transmitted to the generator, data demands are largely constant. This is further reflected in Table 2, which shows the vast bandwidth savings over the 32-circuit evaluation of our representative programs. In particular, for large, complex circuits, the savings are vast: outsourced AES-128 requires 96.3% less bandwidth, while set intersection of size 128 requires 99.7% less bandwidth than in the non-outsourced evaluation. Remarkably, the edit distance 128 problem requires 99.95%, *over 1900 times less bandwidth*, for outsourced execution. The full table is in our technical report [6].

The takeaway from our evaluation is simple: outsourcing the computation allows for faster and larger circuit evaluation than previously possible on a mobile device. Specifically, outsourcing allows users to evaluate garbled circuits with adequate malicious model security (256 circuits), which was previously not possible on mobile devices. In addition, outsourcing is by far the most efficient option if the bandwidth use of the mobile devices is a principle concern.

7 Evaluating Large Circuits

Beyond the standard benchmarks for comparing garbled circuit execution schemes, we aimed to provide compelling applications that exploit the mobile platform with large circuits that would be used in real-world scenarios. We discuss public-key cryptography and the Dijkstra shortest path algorithm, then describe how the latter can be used to implement a privacy-preserving navigation application for mobile phones.

7.1 Large Circuit Benchmarks

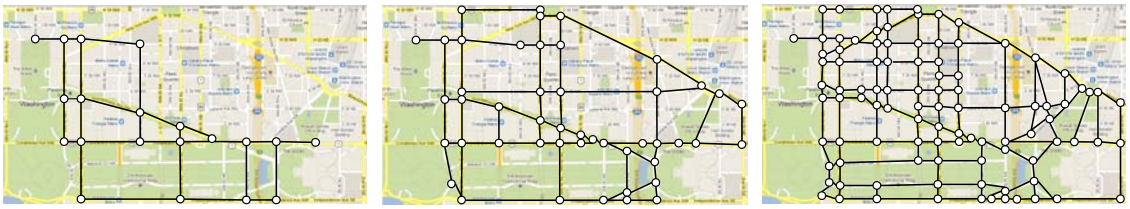
Table 3 shows the execution time required for a blinded RSA circuit of input size 128. For these tests we used a more powerful server with 64 cores and 1 Terabyte of memory. Our testbed is able to give dedicated CPUs when running 32 circuits in parallel. Each circuit would have 1 core for the generation and 1 core for the evaluation. As described in Section 6, larger testbeds capable of executing 128 or 256 cores in parallel would be able to provide similar results for executing the 256 circuits necessary for a 2^{-80} security parameter as they could evaluate the added circuits in parallel. The main difference in execution time would come from the multiple OTs from the mobile device to the outsourced proxy. The RSA circuit has been previously evaluated with *KSS*, but never from the standpoint of a mobile device.

We only report the outsourced execution results, as the circuits are far too large to evaluate directly on the phone. As with the larger circuits described in Section 6, the phone runs out of memory from merely trying to store a representation of the circuit. Prior to optimization, the blinded RSA circuit is $192,537,834$ gates and afterward, comprises $116,083,727$ gates, or 774 MB in size.

The implementation of Dijkstra's shortest-path algorithm results in very large circuits. As shown in Table 3, the pre-optimized size of the shortest path circuit for 20 vertices is $20,288,444$ gates and after optimization is $1,653,542$ gates. The 100-node graph is even larger, with $168,422,382$ gates post optimization, 1124 MB in size. This final example is among the largest evaluated

	32 Circuits Time (ms)	64 Circuits (ms)	128 Circuits (ms)	Optimized Gates	Unoptimized Gates	Size (MB)
RSA128	505000.0 ± 2%	734000.0 ± 4%	1420000.0 ± 1%	116,083,727	192,537,834	774
Dijkstra20	25800.0 ± 2%	49400.0 ± 1%	106000.0 ± 1%	1,653,542	20,288,444	11
Dijkstra50	135000.0 ± 1%	197000.0 ± 3%	389000.0 ± 2%	22,109,732	301,846,263	147
Dijkstra100	892000.0 ± 2%	1300000.0 ± 2%	2560000.0 ± 1%	168,422,382	2,376,377,302	1124

Table 3: Execution time for evaluating a 128-bit blinded RSA circuit and Dijkstra shortest path solvers over graphs with 20, 50, and 100 vertices. All numbers are for outsourced evaluation, as the circuits are too large to be computed without outsourcing to a proxy.



(a) 20 identified intersections. (b) 50 identified intersections. (c) 100 identified intersections.

Figure 8: Map of potential presidential motorcade routes through Washington, DC. As the circuit size increases, a larger area can berepresented at a finer granularity.

garbled circuits to date. While it may be possible for existing protocols to evaluate circuits of similar size, it is significant that we are evaluating comparably massive circuits from a resource-constrained mobile device.

7.2 Privacy-Preserving Navigation

Mapping and navigation are some of the most popular uses of a smartphone. Consider how directions may be given using a mobile device and an application such as Google Maps, without revealing the user's current loca-tion, their ultimate destination, or the route that they are following. That is, the navigation server should remain oblivious of these details to ensure their mutual privacy and to prevent giving away potentially sensitivedetails if the phone is compromised. Specifically, consider plan-ning of the motorcade route for the recent Presidential inauguration. In this case, the route is generally known in advance but is potentially subject to change if sudden threats emerge. A field agent along theroute wants to re-ceivedirections without providing the navigation service any additional details, and without sensitive information about the route loaded to the phone. Moreover, because the threats may be classified, the navigation service does not want the holder of the phone to be given this infor-mation directly. In our example, the user of the phone is trying to determine the shortest path.

To model this scenario, we overlay a graph topology on a map of downtown Washington D.C., encoding in-tersections as vertices. Edge weights are a function of their distance and heuristics such aspotential risks along a graph edge. Figure 8 shows graphs generated based on vertices of 20, 50, and 100 nodes, respectively. Note that the100-nodegraph (Figure8c) encompasses alarger area and provides finer-grained resolution of individual intersections than the 20-node graph (Figure 8a).

Thereisatrade-off between detail and execution time, however; as shown in Table 3, a 20-vertex graph can be evaluated in under 26 seconds, while a 100-vertex graph requires almost 15 minutes with 32 circuits in our 64-core server testbed. The 64 circuit evaluation requires more time: almost 50 seconds for the 20-vertex graph, and almost 22 minutes for a 100-vertex graph. We an-ticipate that based on the role a particular agent might have on aroute, they will beable to generate aroute that covers their particular geographical jurisdiction and thus havean appropriately sized route, with only certain users requiring the highest-resolution output. Additionally, as described in Section 6.3, serverswith moreparallel cores can simultaneously evaluate more circuits, giving faster results for the64 circuit evaluation.

Figure 9 reflects two routes. The first, overlaid with a dashed blue line, is the shortest path under optimal con-ditions that is output by our directions service, based on origin and destination points close to the historical start and end points of the past six presidential inaugural mo-torcades. Now consider that incidents have happened along the route, shown in the figure as a car icon in a hazard zone inside a red circle. The agent recalculates the optimal route, which has been updated by the navi-gation service to assign severe penalties to those corre-sponding graph edges. The updated route returned by the navigation service is shown in the figure as a path with a dotted purple line. In the 50-vertex graph in Fig-ure 8, the updated directions would be available in just over 135 seconds for 32-circuit evaluation, and 196 and ahalf seconds for 64-circuit evaluation.

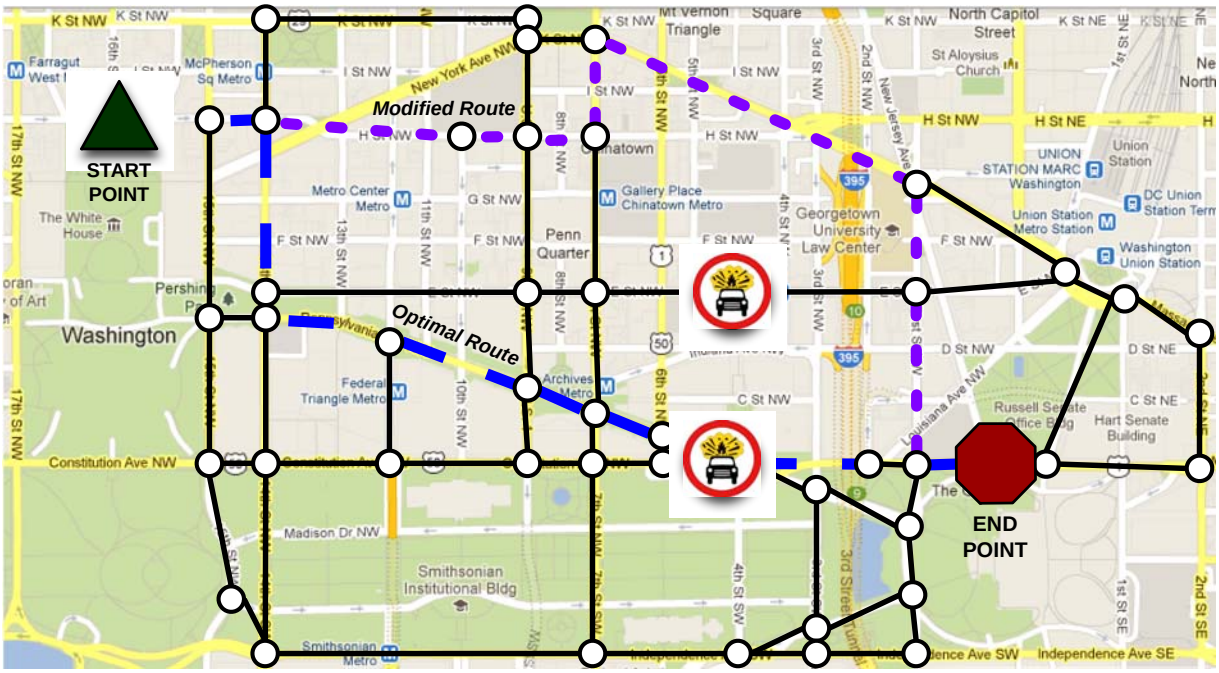


Figure 9: Motorcade route with hazards along the route. The dashed blue line represents the optimal route, while the dotted violet line represents the modified route that takes hazards into account.

8 Conclusion

While garbled circuits offer a powerful tool for secure function evaluation, they typically assume participants with massive computing resources. Our work solves this problem by presenting a protocol for outsourcing garbled circuit evaluation from a resource-constrained mobile device to a cloud provider in the malicious setting. By extending existing garbled circuit evaluation techniques, our protocol significantly reduces both com-putational and network overhead on the mobile device while still maintaining the necessary checks for mali-cious or lazy behavior from all parties. Our outsourced oblivious transfer construction significantly reduces the communication load on the mobile device and can easily accommodate more efficient OT primitivesasthey are developed. The performance evaluation of our protocol shows dramatic decreases in required computation and bandwidth. For the edit distance problem of size 128 with 32 circuits, computation is reduced by 98.92% and bandwidth overhead reduced by 99.95% compared to non-outsourced execution. These savings are illustrated in our privacy-preserving navigation application, which allows a mobile device to efficiently evaluate a massive garbled circuit securely through outsourcing. These results demonstrate that the recent improvements in garbled circuit efficiency can be applied in practical privacy-preserving mobile applications on even the most resource-constrained devices.

Acknowledgments This material is based on research sponsored by DARPA under agreement number FA8750-11-2-0211. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. We would like to thank Benjamin Kreuter, abhi shelat, and Chih-hao Shen for working with us on their garbled circuit compiler and evaluation framework; ChrisPeikert for providing helpful feedback on our proofs of security; Thomas DuBuisson and Galois for their assistance in the performance evaluation; and Ian Goldberg for his guidance during the shepherding process.

References

[1] M. Bellare and S. Micali. Non-interactive oblivious transfer and applications. In *Advances in Cryptology–CRYPTO*, 1990.

[2] M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the annual ACM symposium on Theory of computing*, 1988.

[3] J. Brickell and V. Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *Proceedings of the international conference on Theory and Application of Cryptology and Information Security*, 2005.

[4] R. Canetti, Y. Lindell, R. Ostrovsky, and A. Sahai. Universally composable two-party and multi-party secure computation. In *Proceedings of the annual ACM symposium on Theory of computing*, 2002.

[5] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks (SCN)*, To appear 2013.

[6] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. Technical Report GT-CS-12-09, College of Computing, Georgia Institute of Technology, 2012.

[7] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the annual ACM symposium on Theory of computing*, 1988.

[8] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the "free-xor" technique. In *Proceedings of the international conference on Theory of Cryptography*, 2012.

[9] I. Damgård and Y. Ishai. Scalable secure multiparty computation. In *Proceedings of the annual international conference on Advances in Cryptology*, 2006.

[10] I. Damgård and J. B. Nielsen. Scalable and unconditionally secure multiparty computation. In *Proceedings of the annual international cryptology conference on Advances in cryptology*, 2007.

[11] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Proceedings of the theory and applications of cryptographic techniques annual international conference on Advances in cryptology*, 2008.

[12] M. Green, S. Hohenberger, and B. Waters. Outsourcing the decryption of abe ciphertexts. In *Proceedings of the USENIX Security Symposium*, 2011.

[13] Y. Huang, P. Chapman, and D. Evans. Privacy-Preserving Applications on Smartphones. In *Proceedings of the USENIX Workshop on Hot Topics in Security*, 2011.

[14] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS '12: Proceedings of the 19th ISOC Symposium on Network and Distributed Systems Security*, San Diego, CA, USA, Feb. 2012.

[15] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[16] Y. Huang, J. Katz, and D. Evans. Quid-pro-quotocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[17] A. Iliev and S. W. Smith. Small, stupid, and scalable: Secure computing with faerieplay. In *The ACM Workshop on Scalable Trusted Computing*, 2010.

[18] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Proceedings of the Annual International Cryptology Conference*, 2003.

[19] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[20] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. `http://eprint.iacr.org/`.

[21] S. Kamara, P. Mohassel, and B. Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[22] M. S. Kiraz. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[23] M. S. Kiraz and B. Schoenmakers. A protocol issue for the malicious case of yaos garbled circuit construction. In *Proceedings of Symposium on Information Theory in the Benelux*, 2006.

[24] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free xor gates and applications. In *Proceedings of the international colloquium on Automata, Languages and Programming, Part II*, 2008.

[25] B. Kreuter, a. shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In

Proceedings of the USENIX Security Symposium, 2012.

[26] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2006.

[27] Y. Lindell. Lower bounds and impossibility results for concurrent self composition. *Journal of Cryptology*, 21(2):200–249, 2008.

[28] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2000.

[29] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the annual international conference on Advances in Cryptology*, 2007.

[30] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the conference on Theory of cryptography*, 2011.

[31] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[32] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–a secure two-party computation system. In *Proceedings of the USENIX Security Symposium*, 2004.

[33] Message Passing Interface Forum. The message passing interface (mpi) standard. `http://www.mcs.anl.gov/research/projects/mpi/`, 2009.

[34] P. Mohassel and M. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Proceedings of the Public Key Cryptography conference*, 2006.

[35] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[36] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the annual ACM-SIAM symposium on Discrete algorithms*, 2001.

[37] M. Naor, B. Pinkas, and R. Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the ACM conference on Electronic commerce*, 1999.

[38] N. Nipane, I. Dacosta, and P. Traynor. "Mix-In-Place" anonymous networking using secure function evaluation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[39] C. Peikert, V. Vaikuntanathan, and B. Waters. A framework for efficient and composable oblivious transfer. In *Advances in Cryptology (CRYPTO)*, 2008.

[40] W. Rash. Dropbox password breach highlights cloud security weaknesses. `http://www.eweek.com/c/a/Security/Dropbox-Password-Breach-Highlights-Cloud-Security-Weaknesses-266215/`, 2012.

[41] a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *Proceedings of the Annual international conference on Theory and applications of cryptographic techniques*, 2011.

[42] K. Thomas. Microsoft cloud data breach heralds things to come. `http://www.pcworld.com/article/214775/microsoft_cloud_data_breach_sign_of_future.html`, 2010.

[43] A. C. Yao. Protocols for secure computations. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1982.

PCF: A Portable Circuit Format For Scalable Two-Party Secure Computation

Ben Kreuter
Computer Science Dept.
U. Virginia

Benjamin Mood
Computer and Info. Science Dept.
U. Oregon

abhi shelat
Computer Science Dept.
U. Virginia

Kevin Butler
Computer and Info. Science Dept.
U. Oregon

Abstract

A secure computation protocol for a function $f(x,y)$ must leak no information about inputs x,y during its execution; thus it is imperative to compute the function f in a data-oblivious manner. Traditionally, this has been accomplished by compiling f into a boolean circuit. Previous approaches, however, have scaled poorly as the circuit size increases. We present a new approach to compiling such circuits that is substantially more efficient than prior work. Our approach is based on online circuit compression and lazy gate generation. We implemented an optimizing compiler for this new representation of circuits, and evaluated the use of this representation in two secure computation environments. Our evaluation demonstrates the utility of this approach, allowing us to scale secure computation beyond any previous system while requiring substantially less CPU time and disk space. In our largest test, we evaluate an RSA-1024 signature function with more than 42 billion gates, that was generated and optimized using our compiler. With our techniques, the bottleneck in secure computation lies with the cryptographic primitives, not the compilation or storage of circuits.

1 Introduction

Secure function evaluation (SFE) refers to several related cryptographic constructions for evaluating functions on unknown inputs. Typically, these constructions require an *oblivious* representation of the function being evaluated, which ensures that the control flow of the algorithm will not depend on its input; in the two party case, boolean circuits are most frequently seen. These oblivious representations are often large, with millions and in some cases billions of gates even for relatively simple functions, which has motivated the creation of software tools for producing such circuits. While there has been substantial work on the practicality of secure function evaluation, it was only recently that researchers began investigating the practicality of compiling such oblivious representations from high-level descriptions.

The work on generating boolean circuits for SFE has largely focused on two approaches. In one approach, a library for a general purpose programming language such as Java is created, with functions for emitting circuits [13, 20]. For convenience, these libraries typically include pre-built gadgets such as adders or multiplexers, which can be used to create more complete functions. The other approach is to write a compiler for a high level language, which computes and optimizes circuits based on a high level description of the functionality that may not explicitly state how the circuit should be organized [18, 21]. It has been shown in previous work that both of these approaches can scale up to circuits with at least hundreds of millions of gates on modern computer hardware, and in some cases even billions of gates [13, 18].

The approaches described above were limited in terms of their practical utility. Library-based approaches like HEKM [13] or VMCrypt [20] require users to understand the organization of the circuit description of their function, and were unable to apply any optimizations across modules. The Fairplay compiler [21] was unable to scale to circuits with only millions of gates, which excludes many interesting functions that have been investigated. The poor scalability of Fairplay is a result of the compiler first unrolling all loops and inlining all subroutines, storing the results in memory for later compiler stages. The PALC system [23] was more resource efficient than Fairplay, but did not attempt to optimize functions, relying instead on precomputed optimizations of specific subcircuits. The KSS12 [18] system was able to apply some global optimizations and used less memory than Fairplay, but also had to unroll all loops and store the complete circuit description, which caused some functions to require days to compile. Additionally, the language used to describe circuits in the KSS12 system was

brittle and difficult to use; for example, array index values could not be arbitrary functions of loop indices.

1.1 Our Approach

In this work, we demonstrate a new approach to compiling, optimizing, and storing circuits for SFE systems. At a high level, our approach is based on representing the function to be evaluated as a program that computes the circuit representation of the function, similar to the circuit library approaches described in previous work. Our compiler then optimizes this program with the goal of producing a smaller circuit. We refer to our circuit representation as the *Portable Circuit Format* (PCF).

When the SFE system is run, it uses our interpreter to load the PCF program and execute it. As the PCF program runs, it interacts with the SFE system, managing information about gates internally based on the responses from the SFE system itself. In our system, the circuit is ephemeral; it is not necessary to store the entire circuit, and wires will be deleted from memory once they are no longer required.

The key insight of our approach is that it is not necessary to unroll loops until the SFE protocol runs. While previous compilers discard the loop structure of the function, ours emits it as part of the control structure of the PCF program. Rather than dealing directly with wires, our system treats wire IDs as *memory addresses*; a wire is "deleted" by overwriting its location in memory. Loop termination conditions have only one constraint: they must not depend on any secret wire values. There is no upper bound on the number of loop iterations, and the programmer is responsible for ensuring that there are no infinite loops.

To summarize, we present the following contributions:

- A new compiler that has the same advantages as the circuit library approach
- A novel, more general algorithm for translating conditional statements into circuits
- A new representation of circuits that is more compact than previous representations which scales to arbitrary circuit sizes.
- A portable interpreter that can be used with different SFE execution systems regardless of the security model.

Our compiler is a *back end* that can read the bytecode emitted by a *front end*; thus our compiler allows *any* language to be used for SFE. Instead of focusing on global optimizations of boolean functions, our optimization strategy is based on using higher-level information from the bytecode itself, which we show to be more effective and less resource-intensive. We present comparisons of our compiler with previous work and show experimental results using our compiler in two complete SFE systems, one based on an updated version of the KSS12 system and one based on HEKM. In some of our test cases, our compiler produced circuits only 30% as large as previous compilers starting from the same source code. With the techniques presented in this work, we demonstrate that the RSA algorithm with a real-world key size and real-world security level can be compiled and run in a garbled circuit protocol using a typical desktop computer. To the best of our knowledge, the RSA-1024 circuit we tested is larger than any previous garbled circuit experiment, with more than 42 billion gates. We also present preliminary results of our system running on smartphones, using a modified version of the HEKM system.

For testing purposes, we used the LCC compiler [8] as a front-end to our system. A high-level view of our system, with the LCC front-end, is given in Figure 1.

The rest of this paper is organized as follows: Section 2 is a review of SFE and garbled circuits; Section 3 presents an overview of bytecode languages; Section 4 explains our compiler design and describes our representation; Section 5 discusses the possibility of using different bytecode and SFE systems; Section 6 details the experiments we performed to evaluate our system and results of those experiments; Section 7 details other work which is related to our own; and Section 8 presents future lines of research.

2 Secure Function Evaluation

The problem of secure two-party computation is to allow two mutually distrustful parties to compute a function of their two inputs without revealing their inputs to the opposing party (privacy) and with a guarantee that the output could not have been manipulated (correctness). Yao was the first to show that such a protocol can be constructed for any computable function, by using the *garbled circuits* technique [30]. In his original formulation, Yao proposed a system that would allow users to describe the function in a high level language, which would then be compiled into a circuit to be used in the garbled circuits protocol. The first complete implementation of this design was the Fairplay system given by Malkihi et al. [21].

Oblivious Transfer One of the key building blocks in Yao's protocol is *oblivious transfer*, a cryptographic primitive first proposed by Rabin [25]. In this primitive, the "sender" party holds a database of n strings, and the "receiver" party learns exactly k strings with the guarantee that the sender will not learn which k strings were

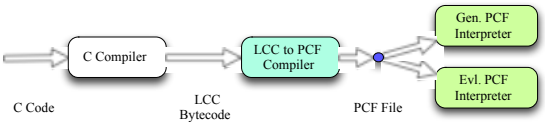


Figure 1: High-level design of our system. We take a C program and compile it down to the LCC bytecode. Our compiler then transforms the LCC bytecode to our new language PCF. Both parties then execute the protocol in their respective role in the SFE protocol. The interpreter could be any execution system.

sent and the receiver will not learn more than k strings; this is known as a k-out-of-n oblivious transfer. Given a public key encryption system it is possible to construct a 1-out-of-2 oblivious transfer protocol [7], which is the building block used in Yao's protocol.

Garbled Circuits The core of Yao's protocol is the construction of garbled circuits, which involves encrypting the truth table of each gate in a circuit description of the function. When theprotocol isrun, thetruth valuesin the circuit will be represented as decryption keys for some cipher, with each gate receiving aunique pair of keysfor itsoutput wire. Thekeysfor agate'sinput wires arethen used to encrypt thekeysfor itsoutput wires. Given asin-gle key for each input wire of the circuit, the party that evaluates the circuit can decrypt a single key that rep-resents a hidden truth value for each gate's output wire, until the output gates are reached. Since this encryption process can beapplied to any circuit, and since any com-putable function has a corresponding circuit family, this allowstheconstruction of asecureprotocol for any computable function.

The typical garbled circuit protocol has two parties though it can be expanded to more. Those two parties are Bob, the generator of the garbled circuit, and Alice, the evaluator of the garbled circuit. Bob creates the gar-bled circuit and therefore knowsthedecryption keys, but doesnot know which specific keysAlice uses. Alice will receivetheinput keysfrom Bob using an obliviousttransfer protocol, and thus learns only one key for each input wire; if the keys are generated independent of Bob's input, Alice will learn only enough to compute the output of the circuit.

Several variations on the Yao protocol havebeen published; a simple description of the garbling and evaluation process follows. Let $f: \{0,1\}^A \tilde{} \{0,1\}^B ! \{0,1\}^j \tilde{} \{0,1\}^k$ be a computable function, which will receive input bits from two parties and produce output bits for each party (not necessarily the same outputs). To garble the circuit, a block cipher hE,D,Gi will be used.

For each wire in the circuit, Bob computes a pair of random keys (k_0,k_1) $(G(1^n),G(1^n))$, which represent logical 0 and 1 values. For each of Alice's outputs, Bob uses these keys to encrypt a 0 and a 1 and sends the pair of ciphertextsto Alice. Bob recordsthekeyscorrespond-ing to his own outputs. The rest of the wires in the cir-cuit areinputs to gates. For each gate, if the truth table is $[v_{0,0},v_{0,1},v_{1,0},v_{1,1}]$, Bob computes the following cipher-text:

$$\tilde{} \begin{bmatrix} E_{k_{l,0}}(E_{k_{r,0}}(k_{v_{0,0}})), E_{k_{l,0}} & (E_{k_{r,1}}(k_{v_{0,1}})) \\ E_{k_{l,1}}(E_{k_{r,0}}(k_{v_{1,0}})), E_{k_{l,1}} & (E_{k_{r,1}}(k_{v_{1,1}})) \end{bmatrix}$$

where $k_{l,}$ and $k_{r,}$ arethekeysfor theleft and right input wires (this can be generalized for gates with more than two inputs). The order of the four ciphertexts is then randomly permuted and sent to Alice.

Now that Alice has the garbled gates, she can begin evaluating the circuit. Bob will send Alice his input wire keys. Aliceand Bob then usean oblivioustransfer to give Alice the keys for her input wires. For each gate, Alice will only be able to decrypt one entry, and will receive one key for the gate's output, and will continue to de-crypt truth table entries until the output wires have been computed. Alice will then send Bob hisoutput keys, and decrypt her own outputs.

Optimizations Numerousoptimizations to thebasic Yao protocol have been published [10, 13, 17, 24, 27]. Of these, themost relevant to compiling circuits isthe "free XOR trick" given by Kolesnikov and Schneider [17]. This technique allows XOR gates to be evaluated without the need to garble them, which greatly reduces the amount of datathat must betransferred and theCPU time required for both thegenerator and theevaluator. Onebasic way to take advantage of this technique is to choose subcircuits with fewer non-XOR gates; Schneider published alist of XOR-optimal circuits for even three-input functions [27].

Huang et al. noted that there is no need for the evaluator to wait for the generator to garble all gates in the circuit [13]. Once a gate is garbled, it can be sent to the evaluator, allowing generation and evaluation to oc-cur in parallel. Thistechnique isvery important for large circuits, which can quickly become too large to store in RAM [18]. Our approach unifies this technique with the use of an optimizing compiler.

3 Bytecode

A common approach to compiler design is to translate a high level language into a sequence of instructions for a simple, abstract machine architecture; this is known as the *intermediate representation* or *bytecode*.Bytecode representations have the advantage of being machine-independent, thus allowing a compiler front-end to be used for multiple target architectures. Optimizations per-

ficing the ability to optimize circuits automatically. Two observations are key to our approach.

Our first observation is that it is possible to free the memory required for storing wire values without computing a reference count for the wire. In previous work, each wire in a circuit is assigned a unique global identifier, and gate input wires are specified in terms of these identifiers (output wires can be identified by the position of the gate in the gate list). Rather than using global identifiers, we observe that wire values are ephemeral, and only require a unique identity until their last use as the input to a gate.

We therefore maintain a table of "active" wire values, similar to KSS12, but change the gate description. In this format, wire values are identified by their index in the table, and gates specify the index of each input wire and an index for the output wire; in other words, a gate is a tuple $\langle t, i_1, i_2, o \rangle$, where t is a truth table, i_1, i_2 are the input wire indexes, and o is the output wire index. When a wire value is no longer needed, its index in the table can be safely used as an output wire for a gate.

Now, consider the following example of a circuit described in the above format, which accumulates the Boolean AND of seven wire values:

$\langle AND_1, 1, 2, 0 \rangle$
$\langle AND_2, 0, 3, 0 \rangle$
$\langle AND_3, 0, 4, 0 \rangle$
$\langle AND_4, 0, 5, 0 \rangle$
$\langle AND_5, 0, 6, 0 \rangle$
$\langle AND_6, 0, 7, 0 \rangle$

Our second observation is that circuits such as this can be described more compactly using a loop. This builds on our first observation, which allows wire values to be overwritten once they are no longer needed. A simple approach to allowing this would add a conditional branch operation to the description format. This is more general than the approach of PAL, which includes loops but allows only simple iteration. Additionally, it is necessary to allow the loop index to be used to specify the input or output wire index of the gates; as a general solution, we add support for indirection, allowing wire values to be copied.

This representation of Boolean circuits is a bytecode for a one-bit CPU, where the operations are the 16 possible two-arity Boolean gates, a conditional branch, and indirect copy. In our system, we also add instructions for function calls (which need not be inlined at compile time) and handling the parties' inputs/outputs. When the secure protocol is run, a three-level logic is used for wire values: 0, 1, or $\perp$, where $\perp$ represents an "unknown" value that depends on one of the party's inputs. In the case of a Yao protocol, the $\perp$ value is represented by a garbled wire value. Conditional branches are not allowed to depend on $\perp$ values, and indirection operations use a separate table of pointers that cannot computed from $\perp$ values (if such an indirection operation is required, it must be translated into a large multiplexer, as in previous work).

We refer to our circuit representation as the *Portable Circuit Format* or PCF. In addition to gates and branches, PCF includes support for copying wires indirectly, a function call stack, data stacks, and setting function parameters. These additional operations do not emit any gates and can therefore be viewed as "free" operations. PCF is modeled after the concept of PAL, but instead of using predefined sub-circuits for complex operations, a PCF file defines the sub-circuits for a given function to allow for circuit structure optimization. PCF includes lower level control structures compared to PAL, which allows for more general loop structures.

In Appendix A, we describe in detail the semantics of the PCF instructions. Example PCF files are available at the authors' website.

4.2 Describing Functions for SFE

Most commonly used programming languages can describe processes that cannot be translated to SFE; for example, a program that does not terminate, or one which terminates after reading a specific input pattern. It is therefore necessary to impose some limitation on the descriptions of functions for SFE. In systems with domain specific languages, these limitations can be imposed by the grammar of the language, or can be enforced by taking advantage of particular features of the grammar. However, one goal of our system is to allow any programming language to be used to describe functionality for SFE, and so we cannot rely on the grammar of the language being used.

We make a compromise when it comes to restricting the inputs to our system. Unlike model checking systems [2], we impose no upper bound on loop iterations or on recursive function calls (other than the memory available for the call stack), and leave the responsibility of ensuring that programs terminate to the user. On the other hand, our system does forbid certain easily-detectable conditions that could result in infinite loops, such as unconditional backwards jumps, conditional backwards jumps that depend on input, and indirect function calls. These restrictions are similar to those imposed by the Fairplay and KSS12 systems [18, 21], but allow for more general iteration than incrementing the loop index by a constant. Although false positives, i.e., programs that terminate but which contain such constructs are possible, our hypothesis is that useful functions and typical compilers would not result in such instruction sequences, and

formed on bytecodearemachineindependent aswell; for example, dead code elimination is typically performed on bytecode, as removing dead code causes programs to run faster on all realistic machines.

For the purposes of this work, we focus on a commonly used bytecode abstraction, the *stack machine*.In this model, operands must be pushed onto an abstract stack, and operations involvepopping operandsoff of the stack and pushing the result. In addition to the stack, a stack machine has RAM, which is accessed by instructions that pop an address off the stack. Instructions in a stack machine are partially ordered, and are divided into subroutines in which there is a total ordering. In addition to simple operations and operations that interact with RAM, astack machinehasoperations that can modify the *program counter*, a pointer to the next instruction to beexecuted, either conditionally or unconditionally.

At a high level, our system translates bytecode programs for a stack machine into boolean circuits for SFE. At first glance, this would appear to be at least highly inefficient, if not impossible, because of the many ways such an input program could loop. We show, however, that imposing only a small set of restrictions on permissible sequences of instructions enables an efficient and practical translator, without significantly reducing theusability or expressive power of the high level language.

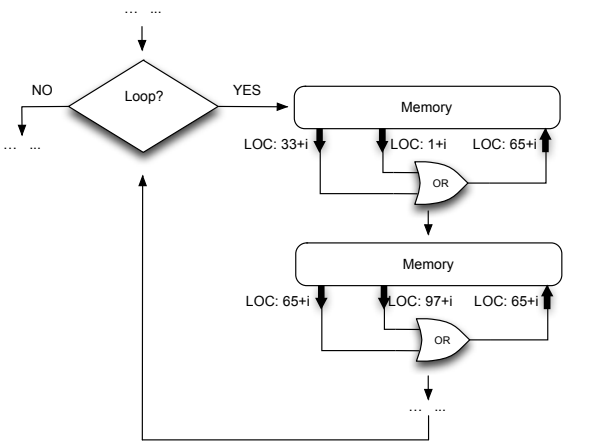


Figure 2: The high-level concept of the PCF design. It is not necessary to unroll loops at compile time, even to perform optimizations on the circuit. Instead, loops can be evaluated at runtime, with gates being computed on-the-fly, and loop indices being updated locally by each party. Wire values are stored in a table, with each gate specifying which two table entries should be used as inputs and where the output should be written; previous wire values in the table can be overwritten during this process, if they are no longer needed.

4 System Design

Our system divides the compiler into several stages, fol-lowing a common compiler design. For testing, we used the LCC compiler front end to parse C source code and produceabytecodeintermediate representation (IR). Our back end performs optimizations and translates the byte-code into a description of a secure computation proto-col using our new format. This representation greatly re-duces thedisk space requirements for largecircuits com-pared to previous work, while still allowing optimiza-tions to be done at the bit level. We wrote our compiler in Common Lisp, using the Steel Bank Common Lisp system.

4.1 Compact Representations of Boolean Circuits

In Fairplay and the systems that followed its design, the common pattern hasbeen torepresent Boolean circuitsas adjacency lists, with each nodein thegraph being agate. The introduces a scalability problem, as it requires stor-age proportional to the size of the circuit. Generating, optimizing, and storing circuits has been a bottleneck for previous compilers, even for relatively simple functions like RSA. Loading such large circuits into RAM is a challenge, as even very high-end machines may not have enough RAM for relatively simple functions.

There have been some approaches to addressing this scalability problem presented in previous work. The KSS12 system reduced the RAM required for protocol executions by assigning each gate's output wire a refer-encecount, allowing thememory used for awirevalueto be deallocated once the gate is no longer needed. How-ever, the compiler bottleneck was not solved in KSS12, as even computing the reference count required memory proportional to thesizeof thecircuit. Even with theengi-neering improvements presented by Kreuter, shelat, and Shen, theKSS12 compiler wasunableto compilecircuits with more than a few billion gates, and required several days to compile their largest test cases [18].

The PAL system [23] also addresses memory requirements, by adding control structures to thecircuit description, allowing parts of the description to be re-used. In the original presentation of PAL, however, alarge circuit file would still be emitted in the Fairplay format when the secure protocol was run. An extension of this work presented by Mood [22] allowed the PAL description to be used directly at runtime, but this work sacrificed the ability to optimize circuits automatically.

Our system builds upon the PAL and KSS12 systems to solve the memory scalability problem without sacri-

we observed no such functions in our experiments with LCC.

4.3 Algorithms for Translating Bytecode

Our compiler reads a bytecode representation of the function, which lacks the structure of higher-level descriptions and poses a unique challenge in circuit generation. As mentioned above, we do not impose any upper limit on loop iterations or the depth of the function call stack. Our approach to translation does not useany symbolic analysis of the function. Instead, we translate the bytecode into PCF, using conditional branches and function callsasneeded and translating other instructions into lists of gates. For testing, we use the IR from the LCC compiler, which is based on the common stack machine model; we will use examples of this IR to illustrate our design, but note that none of our techniques strictly requireastack machine model or any particular features of the LCC bytecode.

In our compiler, we divide bytecode instructions into three classes:

Normal Instructions which have exactly one successor and which can be represented by a simple circuit. Examples of such instructions are arithmetic and bitwise logic operations, operations that push data onto the stack or movedata to memory, etc.

Jump Instructions that result in an unconditional control flow switch to a specific label. This does not include function calls, which we represent directly in PCF. Such instructions areusually used for if/else constructs or preceding the entry to a loop.

Conditional Instructions that result in control flow switching to either alabel or thesubsequent instruction, depending on the result of some conditional statement. Examples include arithmetic comparisons.

In the stack machine model, all operands and the results of operations are pushed onto a global stack. For "normal" instructions, the translation procedure is straightforward: the operands are popped off the stack and assigned temporary wires, the subcircuit for the operation is connected to these wires, and the output of the operation is pushed onto the stack. "Jump" instructions appear, at first, to beequally straightforward, but actually require special care as we describe below.

"Conditional" instructions present achallenge. Conditional jumpswhosetargetsprecedethejump areassumed to beloop constructs, and aretranslated directly into PCF branch instructions. All other conditional jumps require the creation of multiplexers in the circuit to deal with

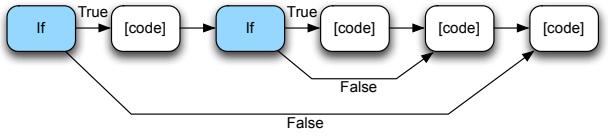


Figure 3: Nested if statements, which can be handled using the stack-based algorithm.

conditional assignments. Therefore, the branch targets must be tracked to ensure that the appropriate condition wires are used to control those multiplexers.

In the Fairplay and KSS12 compilers, the condition wire for an "if" statement is pushed onto a stack along with a "scope" that is used to track the values (wire assignments) of variables. When a conditional block is closed, the condition wire at the top of the stack is used to multiplex the value of all the variables in the scope at the top with the values from the scope second to the top, and then the stack is popped. This procedure relies on the grammar of "if/else" constructs, which ensures that conditional blocks can be arranged as a tree. An example of this type of "if/else" construct is in Figure 3. In a bytecode representation, however, it is possible for conditional blocks to "overlap" with each other without being nested.

In the sequence shown in Figure 4, the first branch's target *precedes* the second branch's target, and indirect loads and assignments exist in the overlapping region of these two branches. The control flow of such an overlap isgiven in Figure5. A stack isno longer sufficient in this case, asthetop of thestack will not correspond to theappropriate branch when the next branch target is encountered. Such instruction sequences are not uncommon in the code generated by production compilers, as they are a convenient way to generate code for "else" blocks and ternary operators.

To handle such sequences, we use a novel algorithm based on a priority queue rather than a stack, and we maintain a global condition wire that is modified as branches and branch targets are reached. When a branch instruction is reached, the global condition wire is updated by logically ANDing the branch condition with the global condition wire. The priority queue is updated with the branch condition and a scope, as in the stack-based algorithm; the priority is the target, with lower targets having higher priority. When an assignment is performed, the scope at the top of the priority queue is updated with the value being assigned, the location being assigned to, the old value, and a copy of the global condition wire. When a branch target is reached, multiplexers are emitted for each assignment recorded in the scope at the top of the priority queue, using the copy of the global condition wire that was recorded. After the

```
EQU4  A
INDIRI4 16
EQU4  B
INDIRI4 24
LABELV  A
ASGNI 4
LABELV  B
ASGNI 4
```

Figure 4: A bytecode sequence where overlapping conditional blocks are not nested; note that the target of the first branch, "A," precedes the target of the second branch, "B."

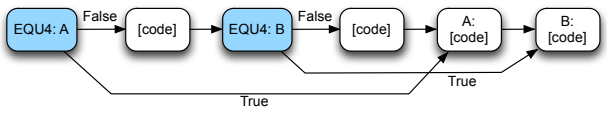


Figure 5: A control flow with overlapping conditional blocks.

multiplexersareemitted, theglobal condition wire isupdated by ORing the *inverse* of the condition wire at the top of the priority queue, and then the top is removed.

Unconditional jumps are only allowed in the forward direction, i.e., only if the jump precedes its target. When such instructionsareencountered, they aretranslated into conditional branches whose condition wire is the inverse of the conjunction of the condition wires of all enclosing branches. In the case of a jump that is not in any conditional block, the condition wire is set to false; this does not necessarily mean that subsequent assignments will not occur, as the multiplexers for these assignments will be emitted and will depend on a global control line that may be updated as part of a loop construct. The optimizer is responsible for determining whether such assignments can occur, and will rewrite themultiplexersas direct assignments when possible.

Finally, it is possible that the operand stack will have changed in the fall-through path of a conditional jump. In that case, the stack itself must be multiplexed. For simplicity, we require that the depth of the stack not change in a fall-through path. We did not observe any such changes to the stack in our experiments with LCC.

4.4 Optimization

One of the shortcomings of the KSS12 system was the amount of time and memory required to perform optimizations on the computed circuit. In our system, optimization isperformed beforeloops areunrolled but after the functionality is translated into a PCF representation. This allows optimizations to be performed on a smaller representation, but increases the complexity of the opti-mization process somewhat.

The KSS12 compiler bases its optimization on a rudimentary dataflow analysis, but without any conditional branches or loops, and with single assignments to each wire. In our system, loops are not eliminated and wires may be overwritten, but conditional branches are eliminated. As in KSS12, we use an approach based on dataflow analysis, but we must make multiple passes to find a fixed point solution to the dataflow equations. Our dataflow equations take advantage of the logical rules of each gate, allowing more gates to beidentified for elimination than the textbook equations identify.

We perform our dataflow analysis on individual PCF instructions, which allowsusto removesinglegateseven whereentirebytecodeinstructions could not beremoved, but which carries the cost of somewhat longer compilation time, on theorder of minutes for theexperiments we ran. Currently, our framework only performs optimization within individual functions, without any interprocedural analysis. Compile times in our system can be reduced by splitting a large procedure into several smaller procedures.

Optimization	128 mult.	5x5 matrix	256 RSA
None	707,244	260,000	904,171,008
Const. Prop.	296,960	198,000	651,504,495
Dead Elim.	700,096	255,875	883,307,712
Both	260,073	131,875	573,156,735

Table 1: Effects of constant propagation and dead code elimination on circuit size, measured with simulator that performs no simplification rules. For each function, the number of non-XOR gatesaregiven for all combinations of optimizations enabled.

4.4.1 Constant Propagation

The constant propagation framework we use is straightforward, similar to the methods used in typical compilers. However, for somegates, simplification rulescan result in constants being computed even when theinputs to a gate are not constant; for example, XORing a variable with itself. The transfer function we use is augmented with acheck against logic simplification rules to account for this situation, but remains monotonic and so convergence is still guaranteed.

4.4.2 Dead Gate Removal

The last step of our optimizer is to remove gates whose output wires are never used. This is a standard bit vector dataflow problem that requires little tailoring for our system. As is common in compilers, performing this step

Function	With	Without	Ratio
16384-bit Comp.	32,228	49,314	65%
128-bit Sum	345	508	67%
256-sit Sum	721	1,016	70%
1024-bit Sum	2,977	4,064	73%
128-bit Mult.	76,574	260,073	20%
256-bit Mult.	300,634	1,032,416	20%
1024-bit Mult.	8,301,962	19,209,120	21%

Table 2: Non-XOR gates in circuits computed by the interpreter with and without the application of simplification rules by the runtime system.

Function	With (s)	Without (s)
16384-bit Comp.	$4.41 \pm 0.3\%$	$4.44 \pm 0.3\%$
128-bit Sum	$0.0581 \pm 0.3\%$	$0.060 \pm 2\%$
256-bit Sum	$0.103 \pm 0.3\%$	$0.105 \pm 0.3\%$
1024-bit Sum	$0.365 \pm 0.3\%$	$0.367 \pm 0.2\%$
128-bit Mult.	$0.892 \pm 0.1\%$	$0.894 \pm 0.1\%$
256-bit Mult.	$3.02 \pm 0.1\%$	$3.04 \pm 0.1\%$
1024-bit Mult.	$39.7 \pm 0.2\%$	$39.9 \pm 0.06\%$

Table 3: Simulator time with simplification rules versus without, using the C interpreter. Times are averaged over 50 samples, with 95% confidence intervals, measured using the *time* function implemented by SBCL.

last yields the best results, as large numbers of gates become dead following earlier optimizations.

4.5 Externally-Defined Functions

Some functionality is difficult to describe well in bytecode formats. For example, the graph isomorphism experiment presented in Section 6 uses AES as a PRNG building block, but the best known description of the AES S-box is given at the bit-level [4], whereas the smallest width operation supported by LCC is a single byte. To compensate for this difficulty, we allow users to specify functions with the same language used internally to translate bytecode operations into circuits; an example of this language is shown in Section 5.1. This allows for possible combinations of our compiler with other circuit generation and optimization tools.

4.6 PCF Interpreter

To use a PCF description of a circuit in a secure protocol, an interpreter is needed. The interpreter simulates the execution of the PCF file for a single-bit machine, emitting gates as needed for the protocol. Loops are not explicitly unrolled; instead, PCF branch instructions are conditionally followed, based on the logic value of some wire, and each wire identifier is treated as an address in memory. This is where the requirement that loop bounds be independent of both parties' inputs is ultimately enforced: the interpreter cannot determine whether or not to take a branch if it cannot determine the condition wire's value.

For testing purposes, we wrote two PCF interpreters: one in C, which is packaged as a reusable library, and one in Java that was used for tests on smartphones. The C library can be used as a simulator or for full protocol execution. As a simulator it simply evaluates the PCF file without any garbling to measure the size of the circuit that would have been garbled in a real protocol. This interpreter was used for the LAN tests, using an updated version of the KSS12 protocol. The Java interpreter was incorporated into the HEKM system for the smartphone experiments, and can also be used in a simulator mode.

4.7 Threat Model

The PCF system treats the underlying secure computation protocol as a black box, without making any assumptions about the threat model. In Section 6, we present running times for smaller circuits in the malicious model version of the KSS12 protocol. This malicious model implementation simply invokes multiple copies of the same PCF interpreter used for the semi-honest version, one for each copy of the circuit needed in the protocol.

4.8 Runtime Optimization

Some optimizations cannot be performed without unrolling loops, and so we defer these optimizations until the PCF program is interpreted. As an example, logic simplification rules that eliminate gates whose output values depend on no more than one of their input wires can only be partially applied at compile time, as some potential applications of these rules might only be possible for some iterations of a loop. While it is possible to compute this information at compile time, in the general case this would involve storing information about each gate for every iteration of every loop, which would be as expensive as unrolling all loops at compile time.

A side effect of applying such logic simplification rules is copy propagation. A gate that always takes on the same value as one of its inputs is equivalent to a copy operation. The application of logic simplification rules to such a gate results in the interpreter simply copying the value of the input wire to the output wire, without emitting any gate. As there is little overhead resulting from the application of simplification rules at runtime, we are able to reduce compile times further by not performing this optimization at compile time.

Function	This Work	KSS12	HFKV
16384 Comp.	32,229	49,149	-
RSA 256	235,925,023	332,085,981	-
Hamming 160	880	-	3,003
Hamming 1600	9,625	-	30,318
3x3 Matrix	27,369	160,949	47,871
5x5 Matrix	127,225	746,177	221,625
8x8 Matrix	522,304	3,058,754	907,776
16x16 Matrix	4,186,368	24,502,530	7,262,208

Table 4: Comparisons between our compiler's output and the output of the KSS12 and Holzer et al. (HFKV) compilers, in terms of non-XOR gates.

For each gate, the interpreter checks if the gate's value can be statically determined, i.e., if its output value does not rely on either party's input bits. This is critical, as some of the gates in a PCF file are used for control flow, e.g., to increment a loop index. Additionally, logic simplification rules are applied where possible in the interpreter. This allows the interpreter to not emit gates that follow an input or which have static outputs even when their inputs cannot be statically determined. As shown in Table 2, we observed cases where up to 80% of the gates could be removed in this manner. Even in a simulator that performs no garbling, applying this runtime optimization not only shows no performance overhead, but actually a very slight performance gain, as shown in Table 3. The slight performance gain is a result of the transfer of control that occurs when a gate is emitted, which has a small but non-trivial cost in the simulator. In a garbled circuit protocol, this cost would be even higher, because of the time spent garbling gates.

5 Portability

5.1 Portability Between Bytecodes

Our compiler can be given a description of how to translate bytecode instructions into boolean circuits using a special internal language. An example, for the LCC instruction "ADDU," is shown in Figure 6. The first line is specific to LCC, and would need to be modified for use with other front-ends. The second line assumes a stack machine model: this instruction reads two instructions from the stack. Following that is the body of the translation rule, which can be used in general to describe circuit components and how the input variables should be connected to those components.

The description follows an abstraction similar to VMCrypt, in which a unit gadget is "chained" to create a larger gadget. It is possible to create chains of chains, e.g., for a shift-and-add multiplier as well. For more complex operations, Lisp source code can be embedded,

```
(``ADDU'' nil second normal nil nil
  (two-stack-arg (x y) (var var)
  (chain [o1 = i1 + i2 + i3,
    o2 = i1 + (i1 + i2) * (i1 + i3)]
  (o2 -> i3
   x -> i1
   y -> i2
   o1 -> stack)
  (0 -> i3))))
```

Figure 6: Code used in our compiler to map the bytecode instruction for unsigned integer addition to the subcircuit for that operation.

which can interact directly with the compiler's internal data structures.

5.2 Portability Between SFE Systems

Both the PCF compiler and the interpreter can treat the underlying secure computation system as a black box. Switching between secure computation systems, therefore, requires work only at the "back end" of the interpreter, where gates are emitted. We envision two possible approaches to this, both of which we implemented for our tests:

1. A single function should be called when a gate should be used in the secure computation protocol. The Java implementation of PCF uses this approach, with the HEKM system.

2. Gates should be generated as if they are being read from a file, with the secure computation system calling a function. The secure computation system may need to provide "callback" functions to the PCF interpreter for copying protocol-specific data between wires. The C implementation we tested uses this abstraction for the KSS12 system.

6 Evaluation

We compiled a variety of functions to test our compiler, optimizer, and PCF interpreter. For each circuit, we tested the performance of the KSS12 system on a LAN, described below. For the KSS12 timings, we averaged the runtime for 50 runs, alternating which computer acted as the generator and which as the evaluator to account for slight configuration differences between the systems. Compiler timings are based on 50 runs of the compiler on a desktop PC with an Intel Xeon 5560 processor, 8GB of RAM, a 7200 RPM hard disk, Scientific Linux 6.3 (kernel version 2.6.32, SBCL version 1.0.38).

Function	Total Gates	non-XOR Gates	Compile Time (s)	Simulator Time (s)
16384-bit Comp.	97,733	32,229	3.40 ± 4%	4.40 ± 0.2%
Hamming 160	4,368	880	9.81 ± 1%	0.0810 ± 0.3%
Hamming 1600	32,912	6,375	11.0 ± 0.4%	0.52 ± 8%
Hamming 16000	389,312	97,175	10.8 ± 0.2%	4.83 ± 0.5%
128-bit Sum	1,443	345	4.70 ± 3%	0.0433 ± 0.4%
256-bit Sum	2,951	721	4.60 ± 3%	0.0732 ± 0.4%
1024-bit Sum	11,999	2,977	4.60 ± 3%	0.250 ± 0.5%
64-bit Mult.	105,880	24,766	71.7 ± 0.2%	0.332 ± 0.4%
128-bit Mult.	423,064	100,250	74.9 ± 0.1%	0.903 ± 0.3%
256-bit Mult.	1,659,808	400,210	79.5 ± 0.9%	3.07 ± 0.2%
1024-bit Mult.	25,592,368	6,371,746	74.0 ± 0.2%	40.9 ± 0.4%
256-bit RSA	673,105,990	235,925,023	381. ± 0.2%	980. ± 0.3%
512-bit RSA	5,397,821,470	1,916,813,808	350. ± 0.2%	7,330 ± 0.2%
1024-bit RSA	42,151,698,718	15,149,856,895	564. ± 0.2%	56,000 ± 0.3%
3x3 Matrix Mult.	92,961	27,369	306. ± 1%	0.256 ± 0.5%
5x5 Matrix Mult.	433,475	127,225	343. ± 0.7%	0.94 ± 2%
8x8 Matrix Mult.	1,782,656	522,304	109. ± 0.1%	3.14 ± 0.3%
16x16 Matrix Mult.	14,308,864	4,186,368	109. ± 0.1%	23.7 ± 0.3%
4-Node Graph Iso.	482,391	97,819	684. ± 0.2%	3.63 ± 0.5%
16-Node Graph Iso.	10,908,749	4,112,135	1040 ± 0.1%	47.0 ± 0.1%

Table 5: Summary of circuit sizes for various functions and the time required to compile and interpret the PCF files in a protocol simulator. Times are averaged over 50 samples, with 95% confidence intervals, except for RSA-1024 simulator time, which is averaged over 8 samples. Run times were measured using the *time* function implemented in SBCL.

Source code for our compiler, our test systems, and our test functions is available at the authors' website.

6.1 Effect of Array Sizes on Timing

Some changes in compile time can be observed as some of the functions grow larger. The dataflow analysis deals with certain pointer operations by traversing the entire local variable space of the function and all global memory, which in functions with large local arrays or programs with large global arrays is costly as it increases the number of wires that optimizer must analyze. Reducing this cost is an ongoing engineering effort.

6.2 Experiments

We compiled and executed the circuits described below to evaluate our compiler and representation. Several of these circuits were tested in other systems; we present the non-XOR gate counts of the circuits generated by our compiler and other work in Table 4. The sizes, compile times, and interpreter times required for these circuits are listed in Table 5. By comparison, we show compile times and circuit sizes using the KSS12 and HFKV compilers in Table 6. As expected, the PCF compiler outperforms these previous compilers as the size of the circuits grow, due to the improved scalability of the system.

Arbitrary-Width Millionaire's Problem As a simple sanity check for our system, we tested an arbitrary-width function for the millionaire's problem; this can be viewed as a string comparison function on 32 bit characters. It outputs a 1 to the party which has the larger input. We found that for this simple function, our performance was only slightly better than the performance of the KSS12 compiler on the same circuit.

Matrix Multiplication To compare our system with the work of Holzer et al. [12], we duplicated some of their experiments, beginning with matrix multiplication on 32-bit integers. We found that our system performed favorably, particularly due to the optimizations our compiler and PCF interpreter perform. On average, our system generated circuits that are 60% smaller. We tested matrices of 3x3, 5x5, 8x8, and 16x16, with 32 bit integer elements.

Hamming Distance Here, we duplicate the Hamming distance experiment from Holzer et al. [12]. Again, we found that our system generated substantially smaller circuits. We tested input sizes of 160, 1600, and 16000 bits.

Integer Sum We implemented a basic arbitrary-width integer addition function, using ripple-carry addition. No

	HFKV			KSS12		
Function	Total Gates	non-XOR gates	Time (s)	Total Gates	non-XOR gates	Time (s)
16384-bit Comp.	330,784	131,103	105. ± 0.1%	98,303	49,154	4.66 ± 0.5%
3x3 Matrix Mult.	172,315	47,871	2.2 ± 4%	424,748	160,949	10.5 ± 0.5%
5x5 Matrix Mult.	797,751	221,625	8.40 ± 0.3%	1,968,452	746,177	48.2 ± 0.2%
8x8 Matrix Mult.	3,267,585	907,776	59.4 ± 0.3%	8,067,458	3,058,754	210 ± 2%
16x16 Matrix Mult.	26,140,673	7,262,208	2,600 ± 7%	64,570,969	24,502,530	2,200 ± 1%
32-bit Mult.	65,121	26,624	6.43 ± 0.3%	15,935	5,983	0.55 ± 5%
64-bit Mult.	321,665	126,529	71.4 ± 0.3%	64,639	24,384	1.6 ± 2%
128-bit Mult.	1,409,025	546,182	999. ± 0.1%	260,351	97,663	6.10 ± 0.6%
256-bit Mult.	5,880,833	2,264,860	16,000 ± 2%	1,044,991	391,935	24.5 ± 0.2%
512-bit Mult.	-	-	-	4,187,135	1,570,303	105. ± 0.2%
1024-bit Mult.	-	-	-	16,763,518	6,286,335	430. ± 0.3%

Table 6: Times of HFKV and KSS12 compilers with circuit sizes. The Mult. program uses a Shift-Add implementation. All times are averaged over 50 samples with the exception of the HFKV 256-bit multiplication, which was run for 10 samples; times are given with 95% confidence intervals.

array references are needed, and so our compiler easily handles this function even for very large input sizes. We tested input sizes of 128, 256, and 1024 bits.

Integer Multiplication Building on the integer addition function, we tested an integer multiplication function that uses the textbook shift-and-add algorithm. Unlike the integer sum and hamming distance functions, the multiplication function requires arrays for both input and output, which slows the compiler down as the problem size grows. We tested bit sizes of 64, 128, 256, and 1024.

RSA (Modular Exponentiation) In the KSS12 system [18], it was possible to compile an RSA circuit for toy problem sizes, and it took over 24 hours to compile a circuit for 256-bit RSA. This lengthy compile time and large memory requirement stems from the fact that all loops are unrolled before any optimization is performed, resulting in a very large intermediate representation to be analyzed. As a demonstration of the improvement our approach represents, we compiled not only toy RSA sizes, but also an RSA-1024 circuit, using only modest computational resources. We tested bit sizes of 256, 512, and 1024.

Graph Isomorpism We created a program that allows two parties to jointly prove the zero knowledge proof of knowledge for graph isomorphism, first presented by Goldreich et al. [9]. In Goldreich et al.'s proof system, the prover has secret knowledge of an isomorphism between two graphs, g_1 and g_2. To prove this, the prover sends the verifier a random graph g_3 that is isomorphic to g_1 and g_2, and the verifier will then choose to learn either the $g_1 \rightarrow g_3$ isomorphism or the $g_2 \rightarrow g_3$ isomorphism. We modify this protocol so that Alice and Bob must jointly act as the prover; each is given shares of an isomorphism between graphs g_1 and g_2, and will use the online protocol to compute g_3 and shares of the two isomorphisms.

Our implementation works as follows: the program takes in XOR shares of the isomophism between g_1 and g_2 and a random seed from both participants. It also takes the adjacency matrix representation of g_1 as input by a single party. The program XORs the shares together to create the $g_1 \rightarrow g_2$ isomorphism. The program then creates a random isomorphism from $g_1 \rightarrow g_3$ using AES as the PRNG (to reduce the input sizes and thus the OT costs), which effectively also creates g_3.

Once the random isomorphism $g_1 \rightarrow g_3$ is created, the original isomorphism, $g_1 \rightarrow g_2$, is inverted to get an isomorphism from $g_2 \rightarrow g_1$. Then the two isomorphisms are "followed" in a chain to get the g_2 to g_3 isomorphism, i.e., for the i^{th} instance in the isomorphic matrix, $iso_{2\rightarrow 3}[i] = iso_{1\rightarrow 3}[iso_{2\rightarrow 1}[i]]$. The program outputs shares of both the isomorphism from g_1 to g_3 and the isomorphism from g_2 to g_3 to both parties.

An adjacency matrix of g_3 is also an output for the party which input the adjacency matrix g_1. This is calculated by using g_1 and the $g_1 \rightarrow g_3$ isomorphism.

6.3 Online Running Times

To test the online performance of our new format, we modified the KSS12 protocol to use the PCF interpreter. Two sets of tests were run: one between two computers with similar specifications on the University of Virginia LAN, a busy 100 megabit Ethernet network, and one between two smartphones communicating over a wifi network.

For the LAN experiments, we used two computers running ScientificLinux 6.3, a four core Intel Xeon E5506 2.13GHz CPU, and 8GB of RAM. No time limit on computation was imposed on these machines, so we were able to run the RSA-1024 circuit, which requires a little less than two days. To compensate for slight con-

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		**Evaluator**	
16384-bit Comp.	$99.8 \pm 0.2\%$	$5.63 \pm 0.6\%$	$26.0 \pm 0.6\%$	$79.4 \pm 0.2\%$
Hamming 1600	$9.13 \pm 0.4\%$	$0.64 \pm 4\%$	$2.9 \pm 4\%$	$6.87 \pm 2\%$
Hamming 16000	$91.2 \pm 0.2\%$	$5.67 \pm 0.7\%$	$28. \pm 3\%$	$69. \pm 2\%$
64-bit Mult.	$0.749 \pm 0.3\%$	$0.158 \pm 0.7\%$	$0.409 \pm 0.3\%$	$0.494 \pm 0.6\%$
128-bit Mult.	$2.04 \pm 0.3\%$	$0.52 \pm 1\%$	$1.25 \pm 0.2\%$	$1.31 \pm 0.6\%$
256-bit Mult.	$5.74 \pm 0.5\%$	$1.2 \pm 2\%$	$4.2 \pm 2\%$	$2.7 \pm 3\%$
1024-bit Mult.	$72.7 \pm 0.2\%$	$28. \pm 4\%$	$60. \pm 2\%$	$40. \pm 3\%$
256-bit RSA	$1940 \pm 0.2\%$	$767. \pm 0.7\%$	$1620 \pm 2\%$	$1080 \pm 3\%$
1024-bit RSA	$1.15 \times 10^5 \pm 0.5\%$	$4.4 \times 10^4 \pm 4\%$	$9.5 \times 10^4 \pm 5\%$	$6.5 \times 10^4 \pm 7\%$
3x3 Matrix Mult.	$5.33 \pm 0.4\%$	$0.403 \pm 0.6\%$	$1.45 \pm 0.8\%$	$4.28 \pm 0.6\%$
5x5 Matrix Mult.	$24.4 \pm 0.2\%$	$1.81 \pm 0.4\%$	$6.75 \pm 0.9\%$	$19.5 \pm 0.4\%$
8x8 Matrix Mult.	$100. \pm 0.2\%$	$7.39 \pm 0.4\%$	$26.8 \pm 0.7\%$	$81.1 \pm 0.3\%$
4-node ISO	$10.1 \pm 0.1\%$	$1.05 \pm 0.7\%$	$4.96 \pm 0.3\%$	$6.15 \pm 0.4\%$
16-node ISO	$116. \pm 0.2\%$	$15.7 \pm 0.6\%$	$71.6 \pm 0.3\%$	$60.3 \pm 0.6\%$

Table 7: Total running time, including PCF operations and protocol operations such as oblivious transfer, for online protocols using the PCF interpreter and the KSS12 two party computation system, on two computers communicating over the University of Virginia LAN. With the exception of RSA-1024, all times are averaged over 50 samples; RSA-1024 is averaged over 8 samples. Running time is divided into time spent on computation and time spent on network operations (including blocking).

figuration differences between the two systems, we alternated between each machine acting as the generator and acting as the evaluator.

We give the results of this experiment in Table 7. We note that while the simulator times given in Table 5 are more than half the CPU time measured, they are also on par with the time spent waiting on the network. Non-blocking I/O or a background thread for the PCF interpreter may improve performance somewhat, which is an ongoing engineering task in our implementation.

6.4 Malicious Model Tests

The PCF system is not limited to the semi-honest model. We give preliminary results in the malicious model version of KSS12. These experiments were run on the same test systems as above, using two cores for each party. We present our results in Table 9. The increased running times are expected, as we used only two cores per party. In the case of 16384-bit comparison, the increase is very dramatic, due to the large amount of time spent on oblivious transfer (as both parties have long inputs).

6.5 Phone Execution

We created a PCF interpreter for use with the HEKM execution system and ported it to the Android environment. We then ran it on two Galaxy Nexus phones where one phone was the generator and another phone was the evaluator. These phones have dual core 1.2Ghz processors and were linked over Wi-Fi using an Apple Airport.

6.6 Phone Trials

As seen in Table 8, we were able to run the smaller programs directly on two phones. Since the interpreter executes slower on a phone and what would have taken a week of LAN trials would have taken years of phone time, we did not complete trials of the larger programs. Not all of the programs had output for the generator, allowing the generator to finish before the evaluator. This leads to a noticeable difference in total running time between the two parties.

Mood's work on designing SFE applications for mobile devices [22] found that allocation and deallocation was a bottleneck to circuit execution. This issue was addressed by substituting the standard *BigInteger* type for a custom class that reduced the amount of allocation required for numeric operations, resulting in a four-fold improvement in execution time. The lack of this optimization in our mobile phone experiments may contribute to the reduced performance that we observed.

In future work, we will port the C interpreter and KSS12 system to Android and run the experiment with that execution system. Since overhead appears to be tied to Android's Dalvik Virtual Machine (DVM), running programs natively should reduce overhead and hence re-

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		**Evaluator**	
16384-bit Comp.	163. ± 0.5%	12. ± 3%	142. ± 0.5%	68. ± 1%
128-bit Sum	5.8 ± 8.2%	1. ± 30%	5.6 ± 8%	3. ± 20%
256-bit Sum	7.3 ± 5.0%	1. ± 30%	6. ± 5%	4. ± 20%
1024-bit Sum	16. ± 3.1%	2. ± 20%	16. ± 3%	6.4 ± 7%
64-bit Mult.	63.3 ± 0.5%	1. ± 10%	66.3 ± 0.6%	5. ± 10%
128-bit Mult.	257. ± 0.2%	3.8 ± 5%	280. ± 0.3%	12. ± 6%
3x3 Matrix Mult.	76.9 ± 0.4%	12. ± 2%	82.0 ± 0.5%	8.5 ± 4%
5x5 Matrix Mult.	352. ± 0.3%	49. ± 2%	371. ± 0.3%	32. ± 4%
8x8 Matrix Mult.	1,588. ± 0.1%	82. ± 3%	1,550. ± 0.1%	120. ± 1%

Table 8: Execution results from the phone interpreter using the HEKM execution system on two Galaxy Nexus phones. Times are averages of 50 samples, with 95% confidence intervals.

Function	CPU (s)	Network (s)	CPU (s)	Network (s)
	Generator		**Evaluator**	
16384-bit comp.	3900 ± 3%	76 ± 4%	2820 ± 2%	1200 ± 10%
128-bit sum	23. ± 2%	21 ± 2%	33.3 ± 0.5%	11.2 ± 0.2%
256-bit sum	63.0 ± 0.4%	10 ± 20%	49. ± 6%	27. ± 4%
1024-bit sum	260 ± 10%	16 ± 6%	187. ± 2%	100 ± 40%
128-bit mult.	192. ± 0.3%	47.2 ± 0.6%	168. ± 0.4%	70.1 ± 1%
256-bit mult.	637. ± 0.5%	160 ± 1%	577. ± 0.3%	210 ± 2%

Table 9: Online running time in the malicious model for several circuits. Times are averaged over 50 samples, with 95% confidence intervals.

duce the performance differential between the phone and PC environments. Additionally, the KSS12 system uses more efficient cryptographic primitives, potentially further improving performance.

7 Related Work

Compiler approaches to secure two-party computation have attracted significant attention in recent years. The TASTY system presented by Henecka et al. [11] combines garbled circuit approaches with homomorphic encryption, and includes a compiler that emits circuits that can be used in both models. As with Fairplay and KSS12, TASTY requires functions to be described in a domain-specific language. The TASTY compiler performs optimizations on the abstract syntax tree for the function being compiled. Kruger et al. developed an ordered BDD compiler to test the performance of their system relative to Fairplay [19]. Mood et al. focused on compiling secure functions on mobile devices with the PALC system, which involved a modification to the Fairplay compiler [23].

Recently, a compiler approach based on bounded model checking was present by Holzer et al. [12]. In that work, the CBMC system [5] was used to construct circuits, which were then rewritten to have fewer non-XOR gates. This approach had several advantages over previous approaches, most prominent being that functions could be described in the widely used C programming language, and that the use of CBMC allows for more advanced software engineering techniques to be applied to secure computation protocols. Like KSS12, however, this approach unrolls all loops (up to some fixed number of iterations), and converts a high level description directly to a boolean circuit which must then be optimized.

In addition to SFE, work on efficient compilers for proof systems has also been presented. Almeida et al. developed a zero-knowledge proof of knowledge compiler for Σ-protocols, which converts a protocol specification given in a domain-specific language into a program for the prover and the verifier to run [1]. Setty et al. presented a system for verifiable computation that uses a modification of the Fairplay compiler, which computes a system of quadratic constraints instead of boolean circuits, and emits executables for the prover and verifier [28, 29]. Our system is somewhat similar to these approaches, in that the circuit representation we present can be viewed as a program that is executed by the par-

ties in the SFE system; however, our approach is unique in its handling of control flow and iterative constructs.

Closely related to our work is the Sharemind system [3, 14], which uses secure computation as a building block for privacy-preserving distributed applications. As in our approach, the circuits used in the secure computation portions of Sharemind are not fully unrolled until the protocol is actually run. Functions in Sharemind are described using a domain-specific language called SecreC. Although there has been work on static analysis for SecreC [26], the SecreC compiler does not perform automatic optimizations. By contrast, our approach is focused on allowing circuit optimizations at the bit-level to occur without having to unroll an entire circuit.

Kerschbaum has presented work on automatically optimizing secure computation at the protocol level, with an approach based on term and expression rewriting [15, 16]. This approach is based on maximizing the use of offline computation by inferring what each party can compute without knowledge of the other party's input, and does not treat the underlying secure computation primitives as a black box. It therefore requires additional work to remain secure in the malicious model. Our techniques could conceivably be combined with Kerschbaum's to reduce the overhead of online components.

8 Future Work

Our compiler can conceivably read any bytecode representation as input; one immediate future direction is to write translations for the instructions of another bytecode format, such as LLVM or the JVM, which would allow functions to be expressed in a broader range of languages. Additionally, we believe that our techniques could be combined with Sharemind, by having our compiler read the bytecode for the Sharemind VM and compute optimized PCF files for cases where garbled circuit computations are used in a Sharemind protocol.

The PCF format does not convey high-level information about data operations or types. Such information may further reduce the size of the circuits that are computed. Static analysis of such information by compilers has been widely studied, and it is possible that our compiler could be extended to support further reductions in the sizes of circuits emitted by the PCF interpreter. High-level information about data structures could also be used to improve the generation of circuits prior to optimization, using techniques recently presented by Evans and Zahur [6].

Our system and techniques can likely be generalized to the multiparty case, and to other representations of functions, such as arithmetic circuits. This would require significant changes to the optimization strategies and goals in our compiler, but fewer changes would be necessary for the PCF interpreter. Similar modifications to support homomorphic encryption systems are also possible.

9 Conclusion

We have presented an approach to compiling and storing circuits for secure computation systems that requires substantially lower computational resources than previous approaches. Empirical evidence of the improvement and utility of our approach is given, using a variety of functions with different circuit sizes and control flow structures. Additionally, we have presented a compiler for secure computation that reads bytecode as an input, rather than a domain-specific language, and have explored the challenges associated with such an approach. We also presented interpreters, which evaluate our new language on both PCs and phones.

The code for the compiler, PCF interpreters, and test cases will be available on the authors' website.

Acknowledgments We would like to thank Elaine Shi for her helpful advice. We also thank Chih-hao Shen for his help with porting KSS12 to use PCF. This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-11-2-0211. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] J. B. Almeida, E. Bangerter, M. Barbosa, S. Krenn, A.-R. Sadeghi, and T. Schneider. A Certifying Compiler For Zero-Knowledge Proofs of Knowledge Based on Σ-Protocols. In *Proceedings of the 15th European conference on Research in computer security*, ESORICS'10, pages 151–167, Berlin, Heidelberg, 2010. Springer-Verlag.

[2] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic Model Checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '99, pages 193–207, London, UK, UK, 1999. Springer-Verlag.

[3] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08*, 2008.

[4] J. Boyar and R. Peralta. A New Combinational Logic Minimization Technique with Applications to Cryptology. In P. Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer Berlin / Heidelberg, 2010.

[5] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[6] D. Evans and S. Zahur. Circuit structures for improving efficiency of security and privacy tools. In *IEEE Symposium on Security and Privacy (to appear)*, 2013.

[7] S. Even, O. Goldreich, and A. Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, June 1985.

[8] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[9] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, July 1991.

[10] V. Goyal, P. Mohassel, and A. Smith. Efficient Two Party and Multi Party Computation Against Covert Adversaries. In *Proceedings of 27th annual international conference on Advances in cryptology*, EUROCRYPT'08, pages 289–306, Berlin, Heidelberg, 2008. Springer-Verlag.

[11] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. TASTY: Tool for Automating Secure Two-partY computations. In *ACM Conference on Computer and Communications Security*, 2010.

[12] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure Two-Party computations in ANSI C. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 772–783, New York, NY, USA, 2012. ACM.

[13] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security Symposium*, 2011.

[14] R. Jagomägis. SecreC: a Privacy-Aware Programming Language with Apllications in Data Mining. Master's thesis, University of Tartu, 2010.

[15] F. Kerschbaum. Automatically optimizing secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 703–714, New York, NY, USA, 2011. ACM.

[16] F. Kerschbaum. Expression rewriting for optimizing secure computation. In *Conference on Data and Application Security and Privacy*, 2013.

[17] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In L. Aceto, I. Damgård, L. Goldberg, M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *ALP 2008*, volume 5126 of *LNCS*, pages 486–498. Springer, 2008.

[18] B. Kreuter, A. Shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 14–14, Berkeley, CA, USA, 2012. USENIX Association.

[19] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*, Alexandria, VA, Oct. 2006.

[20] L. Malka. VMCrypt: modular software architecture for scalable secure computation. In *ACM Conference on Computer and Communications Security*, pages 715–724, 2011.

[21] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay: A Secure Two-Party Computation System. In *13th Conference on USENIX Security Symposium*, volume 13, pages 287–302. USENIX Association, 2004.

[22] B. Mood. Optimizing Secure Function Evaluation on Mobile Devices. Master's thesis, 2012, University of Oregon.

[23] B. Mood, L. Letaw, and K. Butler. Memory-Efficient Garbled Circuit Generation for Mobile Devices. In *Financial Cryptography and Data Security*, volume 7397. Springer Berlin Heidelberg, 2012.

[24] B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure Two-Party Computation Is Practical. In M. Matsui, editor, *Asiacrypt*, volume 5912 of *LNCS*, pages 250–267. Springer, 2009.

[25] M. Rabin. How to Exchange Secrets by Oblivious Transfer. Technical Report TR-81, Harvard Aiken Computation Laboratory, 1981.

[26] J. Ristioja. An analysis framework for an imperative privacy-preserving programming language. Master's thesis, Institute of Computer Science, University of Tartu, 2010.

[27] T. Schneider. *Engineering Secure Two-Party Computation Protocols - Design, Optimization, and Applications of Efficient Secure Function Evaluation*. Springer, 2012.

[28] S. Setty, R. McPherson, A. J. Blumberg, and M. Walfish. Making Argument Systems for Outsourced Computation Practical (Sometimes). In *NDSS*, 2012.

[29] S. Setty, V. Vu, N. Panpalia, B. Braun, A. J. Blumberg, and M. Walfish. Taking proof-based verified computation a few steps closer to practicality. In *Proceedings of the 21st USENIX conference on Security symposium*, Berkeley, CA, USA, 2012.

[30] A. Yao. Protocols for Secure Computations. In *23rd Symposium on Foundations of Computer Science*, pages 160–164. IEEE Computer Society, 1982.

A PCF Semantics

The PCF file format consists of a header section that declares the input size, followed by a list of operations that are divided into subroutines. At runtime, these operations manipulate the internal state of the PCF interpreter, causing gates to be emitted when necessary. The internal state of the PCF interpreter consists of an instruction pointer, a call stack, an array of wire values, and an array of pointers. The pointers are positive integers. Wire values are 0, 1, or $\perp$, where $\perp$ represents a value that depends on input data, which is supplied by the code that invokes the interpreter. Each position in the wire table can be treated as a stack.

Each PCF instruction can take up to 3 arguments. The instructions and their semantics are as follows:

CLABEL/SETLABELC Appears only in the header, used for setting the input size for each party. CLABEL declares the bit width of a value, SETLABELC sets the value.

FUNCTION Denotes the beginning of a subroutine. When the subroutine is called, the instruction pointer is set to the position following this instruction.

GADGET Denotes a branch target

BRANCH Takes two arguments: a target, declared with GADGET, and a location in the wire table. In the wire value is 0, the instruction pointer is set to the instruction following the target. If the wire value is 1, the instruction pointer is incremented. If the wire value is $\perp$, evaluation halts with an error.

FUNC Calls a subroutine, pushing the current instruction pointer onto the call stack.

PUSH Pushes a copy of the wire value at a specified position onto the stack at that position.

POP Pops a stack at a specified position. If there is only one value on that stack, evaluation halts with an error.

ALICEIN32/BOBIN32 Fetches 32 input bits from one party, beginning at a specified *bit* position in that party's input. The bit position is specified by an array of 32 values in the wire table. If any of the values is $\perp$, evaluation halts with an error. The input values will all have the value $\perp$, and will be stored in the wire table at positions 0 through 31.

SHIFT OUT Outputs a single bit for a given party

RETURN Return from a subroutine. The instruction pointer is repositioned to the value popped from the top of the call stack.

STORECONSTPTR Sets a value in the pointer table

OFFSETPTR Adds a value to a pointer, specified by an array of 32 wire values starting at a position in the wire table. If any value in the array is $\perp$, evaluation halts with an error.

PTRTOWIRE Saves a pointer value as a 32 bit unsigned integer. Each of the bits is pushed onto the stack at a location in the wire table.

PTRTOPTR Copies a value from one position in the pointer table to another.

CPY121 Copy a wire value from a position specified by a pointer to a statically specified position.

CPY32 Copy a wire value from a statically specific position to a position specified by a pointer.

$g_{0,0}g_{0,1}g_{1,0}g_{1,1}$ Compute a gate with the specified truth table on two input values from the wire table, with output stored at a specified position. Logic simplification rules are applied when one or both of the input values is $\perp$. If no simplification is possible, then the output will be $\perp$ and the interpreter will emit a gate. This is used for both local computations such as updating a loop index, and for computing the gates used by the protocol.

A.1 Example PCF Description

Below is an example of a PCF file. It iterates over a loop several times times, XORing the two parties' inputs with a bit from the internal state.

```
GADGET: main
CLABEL ALICEINLENGTH 32
CLABEL BOBINLEGNTH 32
CLABEL xxx 32
SETLABELC ALICEINLENGTH 128
SETLABELC ALICEINLENGTH 128
FUNCTION: main
1111 32 0 0
0000 33 0 0
0000 34 0 0
0000 35 0 0
GADGET: L
0110 36 35 34
0001 35 36 36
0110 36 34 33
0001 34 36 36
0110 36 33 32
0001 33 36 36
ALICEINPUT32 0 0
0001 36 0 0
BOBINPUT32 0 0
0001 37 0 0
0110 38 37 36
0110 39 33 38
SHIFT OUT ALICE 39
BRANCH L 35
RETURN xxx
```

SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2013; **00**:1–11
DOI: 10.1002/sec

RESEARCH ARTICLE

For your phone only: custom protocols for efficient secure function evaluation on mobile devices

Henry Carter[1]*, Chaitrali Amrutkar[1], Italo Dacosta[2], and Patrick Traynor[1]

[1]School of Computer Science, Georgia Institute of Technology
[2]KU Leuven, Belgium

ABSTRACT

Mobile applications increasingly require users to surrender private information, such as GPS location or social networking data. To facilitate user privacy when using these applications, Secure Function Evaluation (SFE) could be used to obliviously compute functions over encrypted inputs. The dominant construction for desktop applications is the Yao garbled circuit, but this technique requires significant processing power and network overhead, making it extremely expensive on resource-constrained mobile devices. In this work, we develop *Efficient Mobile Oblivious Computation* (EMOC), a set of SFE protocols customized for the mobile platform. Using partially homomorphic cryptosystems, we develop protocols to meet the needs of two popular application types: location-based and social networking. Using these applications as comparison benchmarks, we demonstrate execution time improvements of 99% and network overhead improvements of 96% over the most optimized garbled circuit techniques. These results show that our protocols provide mobile application developers with a more practical and equally secure alternative to garbled circuits.

KEYWORDS

Secure Function Evaluation, Partially Homomorphic Encryption, Garbled Circuits, Mobile Applications, Privacy

* **Correspondence**
Henry Carter, School of Computer Science, Georgia Institute of Technology.
E-mail: carter@gatech.edu

Received . . .

1. INTRODUCTION

The confluence of high-speed connectivity and device capability has led to the recent surge in mobile application development. While software common to desktop computing (e.g., word processing, email) exists in this space, the most popular mobile applications often provide services based on a user's current context (e.g., location [1], social interconnections [2], etc.). Such applications allow users to make more informed decisions based on their surroundings. However, these applications also regularly expose sensitive data to potentially untrusted parties.

Cryptographers have long worked to develop mechanisms that allow two parties to compute shared results without exposing either individual's sensitive inputs or requiring assistance from a trusted third-party. Such techniques are referred to as *Secure Function Evaluation* (SFE), and provide a set of powerful primitives for privacy-preserving computation. While garbled circuits have been known for nearly 30 years [3], efficient realizations of such schemes have only become possible recently [4, 5, 6, 7, 8]. However, their use on mobile devices, where the nature of applications are different and the use of context sensitive information is the norm and not the exception, has just begun to be assessed [9]. In the past, special-purpose protocols using partially homomorphic encryption [10, 11, 12, 13] have been developed and optimized for specific SFE applications (e.g. cryptographically verifiable voting). This technique promises significant performance gains, but has yet to be applied to mobile applications.

In this paper, we develop custom protocols designed to perform privacy-preserving versions of operations commonly found in applications running on mobile phones. Our *Efficient Mobile Oblivious Computation* (EMOC) techniques use partially homomorphic cryptosystems to restate secure computation as a series of simple arithmetic operations over encrypted inputs. Specifically, we design and implement two privacy-preserving protocols and demonstrate their use in two popular applications:

Prepared using secauth.cls [Version: 2010/06/28 v2.00]

location-based Twitter feeds (a geographic proximity test) and a social networking tool to identify nearby "friends of friends" (a private set intersection). Comparing these applications with equivalent garbled circuit constructions, we demonstrate that our applications can produce the same results at computational and bandwidth costs *reduced by orders of magnitude in some cases.* In so doing, we make the following contributions:

- **Design privacy-preserving mobile applications replacing garbled circuit constructions with partially homomorphic cryptographic primitives:** We design custom privacy-preserving protocols to meet the specific resource constraints of the mobile platform. We then implement these protocols in applications representative of two of the most popular mobile application classes: *location-based messaging* and *social networking*. We prove that our applications provide equivalent security guarantees to their SFE-based counterparts.
- **Propose canonical evaluation tests for mobile SFE applications:** In the desktop world, canonical tests for SFE efficiency have existed for several years. The existence of this common frame of reference for performance between varying techniques has fostered significant growth in the number of schemes available and the performance efficiencies of those schemes. However, these desktop applications are not representative of the types of privacy-preserving computation that would be most useful on the mobile platform (e.g. it is unlikely two mobile users need to securely compute AES). As no such representative test applications have been developed for the mobile platform, we propose a set of test applications to facilitate further study in developing efficient mobile SFE. We will soon open `www.foryourphoneonly.org` as a common repository for mobile SFE applications, providing the research community with a set of existing mobile SFE techniques to compare new techniques as they are developed.
- **Characterize SFE mobile performance profiles:** The relative performance capabilities of garbled circuits on the mobile platform is largely unknown up to this point. In this work, we use our proposed test applications to conduct an extensive performance analysis of five well-known SFE compilers on the Android mobile platform to determine their feasibility in practice. The development of efficient custom privacy-preserving protocols has received significant attention in the past [13, 14, 15, 16, 17, 10]. However, as Huang et al. [7] claim that general circuit compilers provide equivalent performance to the best of these custom protocols, we focus our performance evaluation on demonstrating the performance gains of using custom protocols instead of garbled circuits, and leave the evaluation of existing custom protocols for later work. We demonstrate that our custom-designed SFE protocols offer improvements in execution time as high as 99% and network overhead improvements as high as 96% over the most optimized garbled circuit techniques. Moreover, we examine several garbled circuit optimizations that have never been compared on any platform [5, 7, 8, 18], providing a set of test data to build on in future mobile SFE research.

Our research demonstrates that the performance gains achievable through partially homomorphic constructions merit custom protocols for certain functions. Moreover, our results call for the reevaluation of the recent claims made by Huang et al. [19] that general circuit compilers provide comparable efficiency to custom protocols - *we show empirically and rigorously that this claim does not hold for functions representing the most common applications on the resource-constrained mobile platform.*

2. RELATED WORK

With the development of the "garbled circuit" SFE protocol, Yao demonstrated the possibility of two peer users computing a function without exposing their private inputs [3]. In 2004, Malkhi et al. produced the first practical implementation of Yao garbled circuits in the program Fairplay [4]. Fairplay provided a high-level language and compiler for building the logical circuits that are used to compute functions securely. Fairplay offers the same privacy guarantees as the trusted third party model without requiring an actual third party. Building upon the Fairplay compiler, several techniques have been developed to optimize the generation and evaluation of garbled circuits for various applications [5, 7, 8, 18, 20, 21]. Even with these performance improvements, and considering the assertion of Kerschbaum et al. that communication overhead is of little importance in secure computation [22], garbled circuits are likely to be too expensive for the hardware constraints of mobile devices. Huang et al. began exploring this question in a work examining the performance of pipelined circuits on mobile phones [9]. We thoroughly evaluate this question in our work.

One possible solution to this problem lies in the relatively young area of homomorphic encryption. Henecka et al. demonstrated that homomorphic encryption can be used in conjunction with garbled circuits to provide performance improvements for some SFE problems [6]. However, many special-purpose protocols have been designed to use only partially homomorphic encryption to preserve privacy in applications such as private set intersection [10, 14, 13], voting applications [11], and distributed location privacy [12]. In addition, several protocols for private information retrieval [23] and private stream search [24, 25] leverage this partially homomorphic property of certain encryption schemes. The benefit of the currently available partially homomorphic encryption schemes is that they are efficient, even on mobile

devices [26]. Considering the extreme processing and memory constraints found on the mobile platform, a new set of custom protocols developed for the mobile platform is necessary. While Huang challenges this notion [19], our paper presents privacy-preserving protocols that demonstrate the efficiency gains of custom-designing SFE protocols over general garbled circuit compilers on the mobile platform.

3. CRYPTOGRAPHIC ASSUMPTIONS

Before we define and prove the security of our applications, we specify the requirements for the underlying primitives. We also state basic assumptions that are necessary for the security of our protocols to hold.

3.1. Homomorphic Cryptosystem

The main tool our protocols use in guaranteeing the privacy of all inputs is the homomorphic property of certain cryptosystems. Specifically, we require that any encryption scheme used be multiplicatively homomorphic. That is, given two encryptions of two values, the product of the ciphertexts is equivalent to an encryption of the product of the underlying values. We also require that a ciphertext can be exponentiated by a constant integer such that the result is an encryption of the original plaintext exponentiated by the same integer. Finally, we assume any encryption scheme is semantically secure (i.e., IND-CPA secure) by the following definition from Goldreich [27] (and equivalently Odelu et al. [28]).

Definition 1
Let $X_n^{(i)}, Y_n^{(i)}$ be any messages from the message space of an encryption scheme (G, E, D), Z_n be arbitrary information about the message ensembles X_n, Y_n, and $G_1(1^n)$ be the public encrypting key output by the generation algorithm G. This encryption scheme has uniformly indistinguishable encryptions in the public-key model if for every two polynomials t, l, every probabilistic polynomial-time distinguisher D', every polynomial-time constructible ensemble $T \stackrel{def}{=} \{\overline{T}_n = \overline{X}_n\overline{Y}_n Z_n\}_{n\in\mathbb{N}}$, with $\overline{X}_n = (X_n^{(1)}, ..., X_n^{(t(n))})$, $\overline{Y}_n = (Y_n^{(1)}, ..., Y_n^{(t(n))})$, and $|X_n^{(i)}| = |Y_n^{(i)}| = l(n)$, it holds that

$$|Pr[D'(1^n, Z_n, G_1(1^n), \overline{E}_{G_1(1^n)}(\overline{X}_n)) = 1] - Pr[D'(1^n, Z_n, G_1(1^n), \overline{E}_{G_1(1^n)}(\overline{Y}_n)) = 1]| < \frac{1}{p(n)} \quad (1)$$

for every positive polynomial p and all sufficiently large n's. Here, $\overline{E}_{G_1(1^n)}(\overline{X}_n)$ denotes the set of ciphertexts output by an encryption oracle for the input values in $\overline{X}_n$. The probability is taken over $\overline{T}_n = \overline{X}_n\overline{Y}_n Z_n$ as well as over the internal coin tosses of the relevant algorithms.

We instantiate such an encryption scheme using ElGamal, where the message and ciphertext spaces $\mathcal{M}, \mathcal{C} = G$, where G is a prime-order group. We use a prime-order group because the Decisional Diffie-Hellman (DDH) problem, defined as follows, is known to be hard over prime-order groups. We take our definition from Katz and Lindell [29].

Definition 2
Let G be a group, q be the order of G, g be a generator of G, and x, y, z be exponents in $\mathbb{Z}_q$. The DDH problem is hard for the group G if for all probabilistic polynomial-time distinguishers D' there exists a positive polynomial p such that

$$|Pr[D'(G, q, g, g^x, g^y, g^z) = 1] - Pr[D'(G, q, g, g^x, g^y, g^{xy}) = 1]| \leq \frac{1}{p(n)} \quad (2)$$

3.2. Threat Model

We assume that all privacy guarantees in Section 5 hold against a semi-honest adversary. This means that an adversary will follow the protocol as written, using valid inputs, but will attempt to learn as much as possible outside the jointly computed results by studying logs of all communications [30]. Since this protocol is meant to guarantee the privacy of inputs, we can do nothing if the user chooses false inputs designed to corrupt the protocol. Many garbled circuit implementations also makes this same assumption, *proving security based on semi-honest adversaries* [4, 5, 6, 7]. Our protocols developed under this threat model will provide a foundation for building protocols that can guarantee privacy against other adversarial models.

4. EMOC APPLICATION PROTOCOLS

In this section, we describe in detail the EMOC protocols, applied in two sample applications. We first present a protocol for geographic proximity testing in a Location-Based Twitter application, which allows Alice to subscribe to Bob's tweets without either party revealing their location. Second, we present a private set intersection protocol in our Social Graph connectivity tool, which allows Alice and Bob to determine where their social networks overlap without exposing the identities of all of their friends - an application with potential use when meeting new (and untrusted) people. As a simplified proof of concept, we develop a protocol for solving the canonical millionaire's problem in the technical report version of this work [31].

4.1. Geographic Proximity Test

Location-based messaging, especially for advertisements, has recently received significant attention. Beyond advertising based on location, it offers the potential for useful applications such as a proximity test to alert two people if they are close enough to arrange a meeting. It

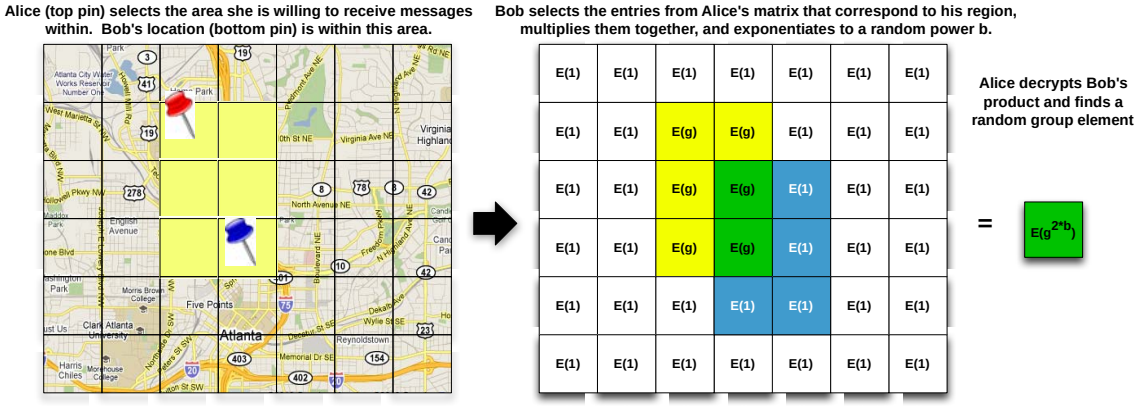


Figure 1. Proximity Test Protocol. We denote $Enc_{pk}(\cdot)$ as $E(\cdot)$. Alice builds a location matrix with encryptions of '1' in every entry except those that correspond to the area she is willing to receive tweets within. In her travel area, she enters encryptions of generator 'g'. Bob selects the entries that correspond to his travel area, multiplies them together, exponentiates by a random blind, and returns the product to Alice. When Alice decrypts, she knows that: if the value is not '1', Bob's tweet is relevant to her. Else, Bob's tweet is irrelevant to her location.

could also be combined with applications like Twitter to allow for location-based tweet filtering and following. However, these applications must query the physical location of a user, which could compromise the user's privacy. To resolve this information leakage, we present a protocol for securely computing when two users are within a chosen proximity of one another. While used in a specific application here, the protocol can be used in any location-based mobile application. The ability to specify an input region of any shape or size allows the proximity test to provide a result at any desired granularity, from the same building to the samecity.

Problem Definition Assume two Twitter users, a follower Alice and a tweeter Bob. Alice selects as her input an area around her current location where, if Bob tweets close to this area, she wants to receive the tweet. Bob inputs an area around his current location where his tweets are relevant. Both of these areas are defined arbitrarily by the user, meaning they can be of *any chosen size and do not haveto becentered on theuser'sgeographic location*.The goal is to compute whether the area where Alice wishes to receive Bob's tweets intersects with the area where Bob's tweets are relevant.

Protocol Definition
Common input:A matrix L of size M˜N where each cell corresponds to a physical region within the city where Alice and Bob are located. Imagine the matrix as a grid laid over a city map. Each cell has a publicly known correlation to the city location beneath it.
Input of Alice:Asetofmatrixentries$A$ corresponding to her general location in L
Input of Bob:Asetofmatrixentries$B$ corresponding to his general location in L
Cryptographic primitives: An encryption scheme $(Gen(), Enc_{pk}(\cdot), Dec_{sk}(\cdot))$ meeting the requirements in Section 3.

1. Alice generates a public/private keypair $pk, sk = Gen()$.
2. Alice generates a matrix $L_A = L$ .For eachentry e_i $2A, L_A[e_i] = g$,where$g$ is a generator of the message space G. $8e_i$ $/2A, L_A[e_i] = 1$.
3. Alice encrypts each entry of L_A as follows: $8e_i$ 2 $L_A, e_i = Enc_{pk}(e_i)$. AlicesendL_A to Bob.
4. Bob homomorphically combines his inputs from L_A into the single ciphertext Q as follows: $8e_i 2 B$, $Q = e_1$ ˜e_2 ˜...˜$e_{|B|}$.
5. Bob blinds Q by exponentiating the ciphertext by a random integer $b2\{1..|G|\}$, generating the result $R = Q^b$.BobreturnsR to Alice.
6. Alicedecrypts R.If $Dec_{sk}(R) = 1$, Alicesendsan output bit $o = 0$ to Bob, meaning Aliceand Bob are not within a close enough proximity to exchange tweets. If $Dec_{sk}(R) = 6$ 1, Alice sends $o = 1$to Bob, and Alice receivesBob'stweet.

Correctness: If any of the elements in A and B overlap, then the multiplied messages will result in $o = g^n$,where n is the number of overlapping entries multiplied by the random blind b. If no entries overlap, the result will be $o = 1^{|B|^b} = 1$.

4.2. Private Set Intersection

Social networking applications are a popular channel for communicating with a mobile device. However, they are also a potential channel to leak private information about a user's social life. If two mobile users were to meet at a party or conference, one might only want to allow the other into her social network based on the friends they already have in common. However, there is currently no application which allowsthiswithout revealing both users'

entire social graphs. This application offers a means for securely revealing only the friends common to both users while maintaining the privacy of the rest of both social graphs. Again, we couch our protocol in an application that is highly relevant to mobile users. However, the protocol can be used in general to compute the intersection of any two sets without revealing any element outside of the intersection. We stress that while our protocol does have a theoretical complexity of $O(n^2)$, our experimental results show that for practical input sizes, *our protocol still executes faster than any garbled circuit implementations available*. This result illustrates that theoretical efficiency does not always carryover into practical, usably efficient algorithms.

Problem Definition Assume two participants, Alice and Bob, who are both members of a social network. Each participant assigns a subset of the social network members as their friends. Given both Alice and Bob's lists of friends, we wish to compute which members of the social network are friends with both Alice and Bob while keeping the rest of their friend lists private.

Protocol Definition
Common input: A predetermined number of friends N to compare.
Input of Alice: A set of bit strings A corresponding to her friend list.
Input of Bob: A set of bit strings B corresponding to his friend list.
Cryptographic primitives: An encryption scheme $(Gen(), Enc_{pk}(\cdot), Dec_{sk}(\cdot))$ meeting the requirements in Section 3 and a one-way, collision-resistant keyed hash function $H : \{0,1\}^* \times \{0,1\}^k \rightarrow G$, given a random key of length k. We assume this hash produces pseudorandom outputs.

1. Alice generates a random hash key $h \in \{0,1\}^k$ and a public/private key pair $pk, sk = Gen()$.
2. Alice hashes and encrypts each of her friends' into a query array Q_A, where for each entry $e_i \in Q_A, e_i = Enc_{pk}(H(A[i], h))$ for $i \in \{1..N\}$. Alice sends Q_A, pk, h to Bob.
3. Bob hashes each of his friends' names from his input B and finds the multiplicative inverse of the hash in the group G, then encrypts his query Q_B, where $\forall e \in Q_B,\ e_i = Enc_{pk}(H(B[i], h)^{-1})$ for $i \in \{1..N\}$.
4. Bob generates a result array Q_R by homomorphically multiplying each entry of his query with all the entries of Alice's query. $\forall e_i \in Q_R, e_i = Q_A[i\%N] \times Q_B[i/N]$, for $i \in \{1..N^2\}$. Here, $\%$ is short for the modulus operator, and $/$ is short for integer division.
5. Bob blinds each entry in Q_R by exponentiating each entry with a random blind. Bob sets N^2 random blinds as $b_i, i \in \{1..N^2\}$. Bob blinds each entry of Q_R, $\forall e_i \in Q_R, e_i = e_i^{b_i}$ for $i \in \{1..N^2\}$. Bob sends Q_R to Alice.
6. Alice decrypts each entry of Q_R, where $\forall e \in Q_R, e_i = Dec_{sk}(e_i)$. Alice defines the set of matching indices as M such that if $Q_R[i] = 1$ then $i\%N \in M$. Alice returns the output array $o = A[i] \in A$ where $i \in M$.

Correctness: If a string matches in both Alice and Bob's inputs, then the product of that matching string x will be $H(x,h) * H(x,h)^{-1} = 1$. Otherwise, the product will be two pseudorandom elements x, y from G as $(x * y)^{b_i}$.

5. PRIVACY GUARANTEES

In this section, we define our threat model and prove the privacy guarantees of both EMOC protocols. For each protocol, we show two properties: the security of the two-party computation and the amount of information revealed by the result of computation.

5.1. Definitions

In all of our protocols, we assume the standard definition of a semi-honest adversary, described in Lindell and Pinkas' work [30]. Essentially, this states that both parties will follow the protocol as written but will attempt to learn information beyond the computed result from transcripts of the interaction. This assumption is also made by related efforts in this space [4, 5, 6, 7, 30]. To prove a protocol secure against semi-honest adversaries, we use the concept of indistinguishability between Alice's view in a real execution and a simulator's generation in an ideal execution. In the ideal world, two participants A, B send their inputs a, b to a trusted third party which performs some computation and returns the result $f(a,b)$. The proof idea is to show that a simulator S in the ideal world can simulate A's view in the real protocol.

Definition 3
Semi-honest security: For any deterministic functionality $f(x,y)$ and semi-honest parties P_1 and P_2, we say that protocol π securely computes f in the presence of semi-honest adversaries if there exists ppt algorithms S_1 and S_2 such that:

$$S_1(x, f(x,y))_{x,y\in\{0,1\}^*} \overset{c}{\approx} view_1^{\pi}((x,y), output^{\pi}(x,y))_{x,y\in\{0,1\}^*} \quad (3)$$

$$S_2(y, f(x,y))_{x,y\in\{0,1\}^*} \overset{c}{\approx} view_2^{\pi}((x,y), output^{\pi}(x,y))_{x,y\in\{0,1\}^*} \quad (4)$$

5.2. Location Privacy

Theorem 1
Location Privacy: Assuming the encryption scheme used in the proximity test protocol is semantically secure, the

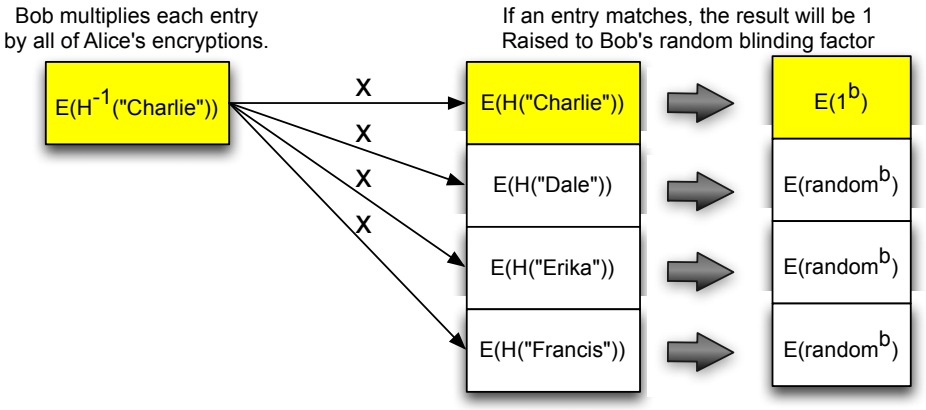


Figure 2. Private set intersection: We denote $Enc_{pk}(\cdot)$ as $E(\cdot)$. Bob homomorphically multiplies each entry in his array by every entry in Alice's array. He then exponentiates by a unique blinding factor for all of the resulting values. Alice receives these values and decrypts them. If an entry is equal to 1, Alice knows there is a match.

proximity test protocol is secure in the presence of semi-honest adversaries.

Proof
We prove the security of the protocol separately for each participant.

When Alice is corrupt, her view of the protocol $view_A((a,b),o)$ consists only of the message R. We run the following protocol for simulator $S_1(A,o)$:

1. S_1 receives L_A from Alice. If $o = 0$, S_1 selects any entry $e \in L_A$ such that $e \notin A$ and sets $Q = e$. Else, S_1 sets $Q = e \in L_A$ such that $e \in A$.
2. S_1 chooses a random integer b', generates $R' = Q^{b'}$, and returns R' to Alice.

Proof
If $o = 0$, the message $Dec_{sk}(R)$ is identical in both the real and simulated execution, implying that $S_1(a,o) \overset{c}{\equiv} view_A((a,b),o)$. If $o = 1$, by the DDH problem in Definition 2, we have that $|Pr[A(G,q,g,g^a,g^b,g^{b'}) = 1] - Pr[A(G,q,g,g^a,g^b,g^{ab}) = 1]| \leq \frac{1}{p(n)}$ for some polynomial p. This implies again that $S_1(a,o) \overset{c}{\equiv} view_A((a,b),o)$. Therefore, the proximity test protocol is secure when Alice is corrupt. □

When Bob is corrupt, his view of the protocol $view_B((a,b),o)$ consists of the messages L_A and o. We run the following protocol for simulator $S_2(B,o)$:

1. S_2 generates L_A' by filling each entry with random values from the ciphertext space C of the encryption scheme and sends L_A' to Bob.
2. When Bob replies with R, S_2 sends o to Bob.

Proof
By the definition of semantic security in Definition 1, $|Pr[B(1^n,Z_n,pk,\overline{E}_{pk}(L_A)) = 1] - Pr[B(1^n,Z_n,pk,\overline{E}_{pk}(L_A')) = 1]| < \frac{1}{p(n)}$ for some polynomial p and arbitrary information Z_n about the plain texts. The final output o is identical in both executions, implying that $S_2(b,o) \overset{c}{\equiv} view_B((a,b),o)$. Therefore, the proximity test protocol is secure when Bob is corrupt. □

Given the existence of simulators S_1, S_2, this proves the theorem. □

5.3. Private Set Intersection Privacy

Theorem 2
Private Set Intersection Privacy: Assuming the encryption scheme used in the private set intersection protocol is semantically secure and that the secure hash function used is pseudorandom and one-way, the private set intersection protocol is secure in the presence of semi-honest adversaries.

Proof
Again, we prove separately the security of our protocol for Alice and Bob.

When Alice is corrupt, her view of the protocol $view_A((a,b),o)$ consists only of the message Q_R. We run the following protocol for simulator $S_1(A,o)$:

1. S_1 receives (Q_A, pk, h) from Alice. S_1 generates Q_B by hashing the names in o using h, finding the multiplicative inverse of each hash in G, and encrypting using pk (as defined in the protocol). If $|o| < N$, S_A fills the remaining entries with a set of hashed, inverted, and encrypted names F such that $\forall e \in F, e \notin A$. S_1 shuffles the entries of Q_B.
2. S_1 performs the homomorphic operations as defined in the protocol, blinds each entry of Q_R' with a random exponent b', and returns Q_R' to Alice.

Proof
By Definition 2, we have for every entry in

$Dec_{sk}(Q'_R[i]) = g^{b'}$ $|Pr[A(G, q, g, g^x, g^b, g^{b'}) = 1] - Pr[A(G, q, g, g^x, g^b, g^{xb}) = 1]| \leq \frac{1}{p(n)}$ for some polynomial p (note here that g^x is the hashed name of one of Bob's friends multiplied by the hashed name of one of Alice's different friends). Thus, we have that $S_1(a, o) \overset{c}{\approx} view_A^{\pi}((a, b), o)$ when the hash function H is one-way, collision-resistant, and pseudorandom.

Therefore, the private set intersection protocol is secure when Alice is corrupt. □

When Bob is corrupt, his view of the protocol $view_B^{\pi}((a, b), o)$ consists of the messages Q_A and o. We run the following protocol for simulator $S_2(B, o)$:

1. S_2 generates Q'_A by filling an array of length N with elements from the ciphertext space $\mathcal{C}$ of the encryption scheme. S_2 generates a random public key pk and a random hash key h and sends (Q'_A, pk, h), to Bob.
2. When Bob replies with Q_R, S_2 sends o to Bob.

Proof
By Definition 1, we have that $|Pr[B(1^n, Z_n, pk, \overline{E}_{pk}(Q_A) = 1] - Pr[B(1^n, Z_n, pk, \overline{E}_{pk}(Q'_A)) = 1]| < \frac{1}{p(n)}$ for some polynomial p and arbitrary information Z_n about the plain texts. The distributions of pk and h in both executions are identical. The final output o is identical in both executions, implying that $S_2(b, o) \overset{c}{\approx} view_B^{\pi}((a, b), o)$. Therefore, the private set intersection protocol is secure when Bob is corrupt. □

Given the existence of simulators S_1, S_2, this proves the theorem. □

6. PERFORMANCE ANALYSIS

While developing new SFE protocols is useful, the main contribution of our work is the establishment of an efficient technique for performing SFE on the mobile platform. Another contribution is our canonical test set that will facilitate future comparisons between techniques and encourage additional development of efficient mobile SFE schemes. Using these tests, our results demonstrate that through custom designed protocols, we can take advantage of optimizations that are not available to general-purpose garbled circuit compilers. This allows our protocols to execute in time that would be usable by the average mobile user. We also provide baseline statistics that compare a variety of garbled circuit techniques that have yet been untested on the mobile platform. *We note that we do not evaluate any other custom protocols as Huang et al. [19] claims to have equivalent performance.*

6.1. Mobile SFE Benchmarking Applications and Metrics

To demonstrate the efficiency of a given secure function evaluation technique, we chose two protocols that are widely applicable in mobile applications: a geographic proximity test and a private set intersection protocol. As we have already shown, these particular functions would be very useful in some of the most popular mobile applications. As such, they are a critical benchmark when examining new mobile SFE techniques.

In addition to presenting these test applications, we propose a set of metrics to compare efficiency between techniques executing the test applications. The first is average execution time, taken over 10 executions with 95% confidence in the error margin. To demonstrate feasibility for practical use, we use inputs that correspond to real values a user might present to such an application. We found that the execution time for large input sizes tended to be the limiting factor in performance rather than memory usage. If an input size caused the application to exceed the 24 MB automatically allotted by the Android OS, we increased the heap size and retried the experiment. Even when allotted extra memory, these executions still failed to complete. So, if an input exceeded the standard allocation of 24 MB, we considered it unable to complete. The second metric is total network usage between both parties, measured as the number of bytes exchanged during the protocol. To capture the amount of data exchanged, we used the "Shark for Root" Android application to capture network traffic [32], then examined the data in Wireshark [33]. As mobile devices are often required to function with costly network connections (both in terms of energy and billing), minimizing the amount of traffic required between two parties is critical to efficient performance.

To demonstrate the practicality of our protocols in comparison to existing garbled circuit compilation techniques, we perform the experiments defined above for our scheme as well as five other garbled circuit compilation techniques. We selected these techniques because they represent the full range of general garbled circuit compilers available. We opted not to include the TASTY framework by Henecka et al. [6] in this work, as this framework requires circuits to be constructed and optimized by hand on a per-function basis. The focus of our work is to show the performance benefits of our scheme against garbled circuit compilers designed to compile *any general secure function* from a higher level language.

The first garbled circuit technique we evaluate is Fairplay [4], the standard desktop implementation of Yao's garbled circuit technique. Next, we examine Kruger's Ordered Binary Decision Diagram optimization [5], which produces smaller, more efficient garbled circuits through a modified representation. Third, the Parallelized scheme by Kreuter et al. [18] incorporates a number of optimizations for evaluating large circuits in parallel on server-class machines. The fourth scheme we evaluate is the Pseudo-Assembly Language (PAL) Compiler by Mood et al. [8], which produces circuits in a memory-efficient manner using an intermediate circuit compiler language. Finally, we examine the pipelined evaluation technique of Huang

Protocol	Input size	SFE scheme	Avg. exec. time (sec.)	Network use (KB)
Proximity Test	500 cells	EMOC	0.0165 (± 0.0001)	128.256
		OBDD	23.1480 (± 0.0351)	1,765.764
		Parallelized	26.2353 (± 0.0836)	1,854.049
		PAL	35.1888 (± 0.0487)	2,029.439
		Pipelined	11.1293 (± 0.0332)	603.497
		Fairplay	NA	NA
Private Set Intersection	20 friends	EMOC	3.7466 (± 0.0042)	107.520
		OBDD	124.4921 (± 0.2809)	2,879.016
		Parallelized	107.8990 (± 0.4249)	2,669.284
		PAL	130.7570 (± 0.2013)	3,025.966
		Fairplay	NA	NA
	16 friends	Pipelined	45.7061 (± 0.1254)	3,401.133

Table I. Our experimental results. Values are present for the maximum input size measure across all applications for accurate comparison. In the private set intersection protocol, the Pipelined execution environment required input size to be a power of two.

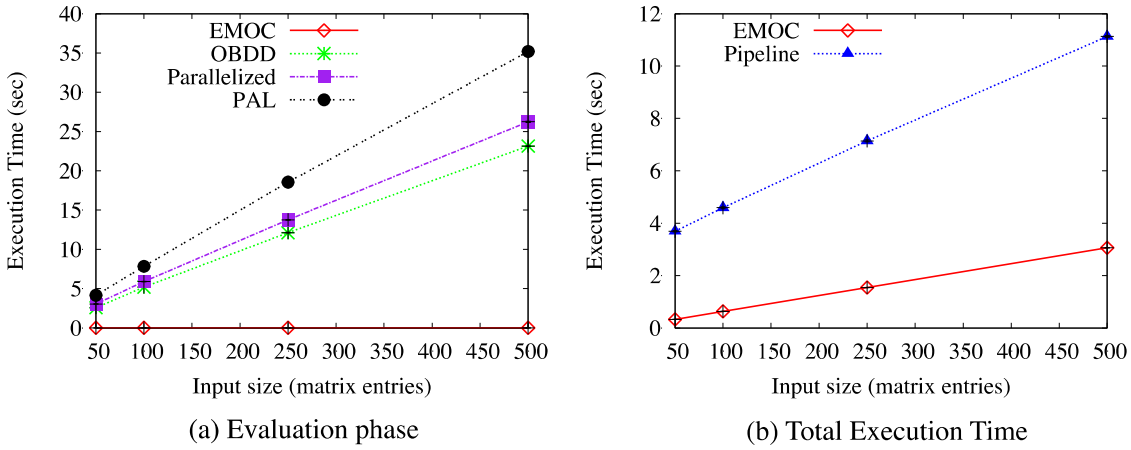


Figure 3. Proximity test execution times. Note that for the online execution, for all input sizes our application runs in a fixed amount of time while all garbled circuits show increasing execution times with increasing input size.

et al. [7], which splits garbled circuits into layers that can be generated and evaluated separately.

For all of the garbled circuit compilation techniques *except the pipelined evaluation*, we split the scheme into two phases: preprocessing and execution. For the preprocessing phase, we compiled the garbled circuits on a desktop and then examined their evaluation times, the execution phase, on the mobile device. To assure fair representation across techniques, we either compiled the circuits ourselves using compiler framework provided by the technique author or had the author compile the same SFDL circuit description on their own machines. In our own protocols, we consider the time Alice takes to generate her query as the preprocessing phase, while the online communication between Alice and Bob constitutes the evaluation phase. In the case of pipelined circuit evaluation, no such preprocessing phase exists, since the circuits are generated in layers during the online evaluation between the two parties. While this inability to amortize computation costs is a performance weakness when compared to other SFE techniques, we chose to perform separate experiments that compare their pipelining execution time against our combined preprocessing and online execution time. For these experiments, we acquired the same mobile implementation developed by Huang et al. [9] for pipelining the generation and evaluation of garbled circuits on a mobile device. While this does not provide a true picture of the efficiency gains in our amortized execution, it still shows the significant performance gains of using partially homomorphic encryption instead of garbled circuits for mobile SFE applications.

We evaluated all precompiled garbled circuits (i.e. *not* pipelined circuits) on the standard Java Fairplay platform, which we ported to an Android application. Our own protocols were written in C and cross-compiled to run natively using the Android toolchain [34]. We use ElGamal with 1024-bit keys to encrypt and HMAC-md5 to hash. All performance figures were taken on the Samsung Nexus S smartphone.

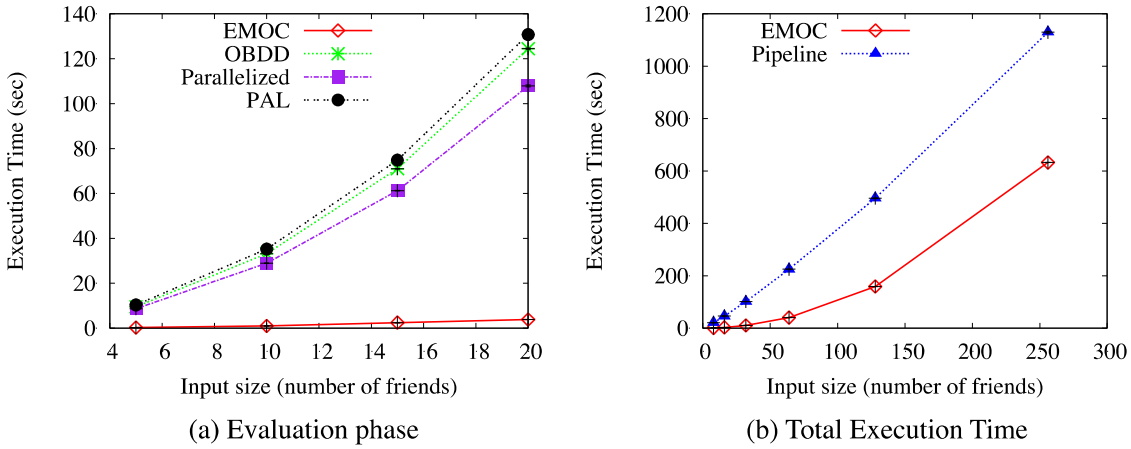


Figure 4. Private set intersection execution times. For every garbled circuit technique except the pipelined circuits, we were only able to run experiments up to inputs of size 20 due to the large memory requirements of Fairplay.

6.2. Results

The results in Table I clearly demonstrate the advantages of our custom protocols over every garbled circuit technique tested. In both test applications, we see a significant increase in execution time and network usage for all garbled circuit implementations, even in the comparison of total execution time in Figures 3b and 4b. Because general purpose compilers cannot take advantage of optimizations inherent to specific functions, they tend to produce circuits with irregular performance profiles. This is clearly seen in Figures 3a and 4a, where garbled circuit techniques do not consistently outperform one another between applications. For example, in the proximity test protocol, the OBDD scheme outperforms the parallelized scheme. However, this ordering is reversed for the private set intersection protocol. Ultimately, these fluctuations in performance are eclipsed by the gains achieved through our custom designed protocols, where online execution times were *at least 96% better* than the fastest garbled circuit technique. In the best case, our techniques reduced execution time by *three orders of magnitude*.

One advantage of our protocols is that for increasing input sizes, our proximity test protocol only requires an increase in preprocessing time, while the online execution remains constant across all input sizes. By contrast, *every* garbled circuit technique showed increasing execution times as input size increased, emphasizing this significant benefit of our customized protocols. In addition, the optimizations that are incorporated into garbled circuit schemes do not consistently provide any benefit on the constrained mobile platform. For example, the highly parallelized scheme performs about as well as non-parallelized garbled circuits, simply because most mobile phone hardware contains single-core processors. In the case of the pipelined evaluation circuits, we see

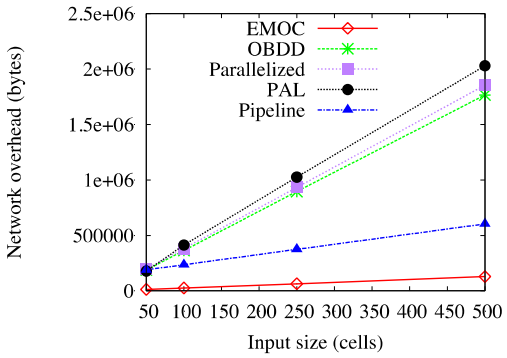


Figure 5. Proximity test network usage. Note that even the most optimized garbled circuit evaluation technique requires over four times the amount of network traffic used by EMOC.

an optimization that solves the problem of memory-intensive preprocessing (i.e., circuit generation), but does not allow for amortized execution time. EMOC provides a solution with an efficient preprocessing phase, where each precomputed ciphertext requires only 256 bytes of memory, *as well as* faster amortized execution. Contrary to Huang et al. [19], these results show that custom protocols significantly outperform the best available garbled circuit optimizations on mobile devices.

6.3. Network Overhead

In addition to faster execution, our protocols significantly reduce the amount of network overhead compared to garbled circuit techniques. For the proximity test protocol (Figure 5), we observed a 78% reduction from the best garbled circuit technique, pipelined execution. In

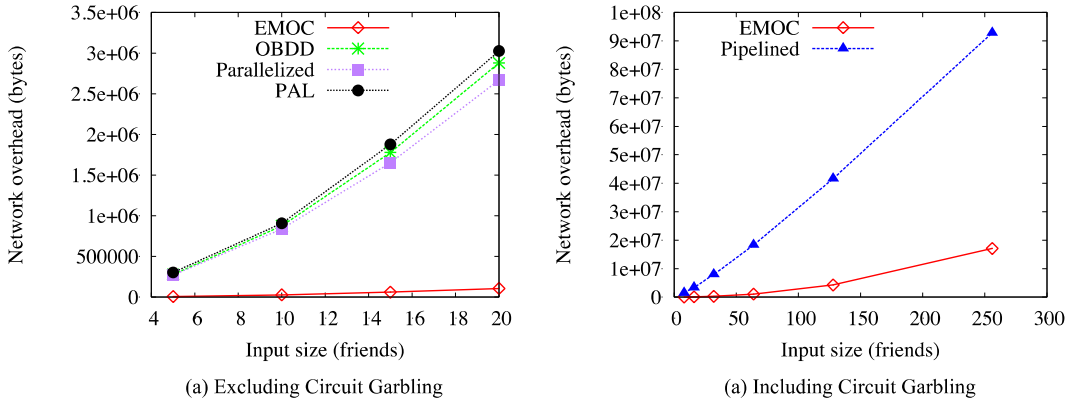


Figure 6. Private set intersection network traffic. For every garbled circuit technique shown in graph (a), we were only able to run experiments up to inputs of size 20 before Fairplay was unable to complete execution. Figure (b) shows the network usage for large inputs in our scheme and the pipelined evaluation scheme. Since the pipelined evaluation scheme implementation of private set intersection only accepts inputs of size 2^k, we show these results separately.

the private set intersection protocol (Figure 6), the improvement swelled to 96% over the best garbled circuit technique, the parallelized circuits. This improvement is due largely to the fact that our protocol does not use oblivious transfers to exchange inputs. In theory, there exist a number of oblivious transfer schemes that perform with efficient ($O(n)$) communication overhead [35]. However, it is clear that these efficiencies do not always carry over in practice. In the case of our private set intersection protocol, we exchange theoretical "efficiency" for practical usefulness by employing techniques that use less network overhead in exchange for complexity that is theoretically less efficient ($O(n^2)$). It is important to note that for all garbled circuit techniques except the pipelined circuit evaluation, these graphs do not include the data required to initially send the circuit from the generating party to the evaluating party. Depending on the application and input size, these circuit files could require as much as 935 KB of additional bandwidth over the measured amounts further decreasing the feasibility of garbled circuit protocols.

7. CONCLUSION

As mobile phones become more popular, new techniques will be needed to protect the private information used in many of their applications. Garbled circuit constructions offer an increasingly realistic solution in the desktop space, but require too much processing power and network overhead to be practical on the mobile platform. By replacing garbled circuits with homomorphic encryption operations, our EMOC protocols demonstrate that certain privacy-preserving functions can be evaluated with great efficiency on the mobile platform. In addition, our canonical test applications provide a common reference point when comparing SFE techniques on the mobile platform. Using these metrics, our performance evaluation demonstrates improvements in our protocols of greater than 99% over the most efficient garbled circuit constructions, as well as an initial characterization of the performance capabilities of several garbled circuit optimizations on the mobile platform. Based on these results, we present our protocols as an efficient method for implementing SFE into some location-based and social networking applications. To foster further research into efficient mobile SFE, we make our test metrics and applications available to the research community at `www.foryourphoneonly.org` and encourage other authors working in this space to post their implementations as well.

ACKNOWLEDGEMENTS

This material is based on research sponsored by DARPA under agreement number FA8750-11-2-0211. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government. We would also like to thank Kevin Butler, Peter Chapman, Yan Huang, Benjamin Kreuter, Louis Kruger, Benjamin Mood, Abhi Shelat and Chih-hao Shen for their assistance in generating garbled circuits under their respective schemes.

REFERENCES

1. Constandache I, Bao X, Azizyan M, Choudhury R. Did you see Bob?: human localization using mobile phones. *Proceedings of the ACM International Conference on Mobile Computing and Networking (Mobicom)*, Chicago, IL, USA, 2010.
2. Banerjee N, Agarwal S, Bahl P, Chandra R. Virtual compass: relative positioning to sense mobile social interactions. *Technical Report*, Microsoft Research 2010. URL http://www.springerlink.com/index/K81H08U2767N2117.pdf.
3. Yao AC. How to generate and exchange secrets. *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Toronto, Canada, 1986.
4. Malkhi D, Nisan N, Pinkas B, Sella Y. Fairplay – A Secure Two-Party Computation System. *Proceedings of the USENIX Security Symposium (SECURITY)*, San Diego, CA, USA, 2004.
5. Kruger L, Jha S, Goh EJ, Boneh D. Secure Function Evaluation with Ordered Binary Decision Diagrams. *Proceedings of the ACM conference on Computer and communications security (CCS)*, Alexandria, VA, USA, 2006.
6. Henecka W, Kögl S, Sadeghi Ar, Schneider T, Wehrenberg I. TASTY : Tool for Automating Secure Two-partY computations. *Proceedings of the ACM conference on Computer and Communications Security (CCS)*, Chicago, IL, USA, 2010.
7. Huang Y, Evans D, Katz J, Malka L. Faster Secure Two-Party Computation Using Garbled Circuits. *Proceedings of the USENIX Security Symposium*, San Francisco, CA, USA, 2011.
8. Mood B, Letaw L, Butler K. Memory-Efficient Garbled Circuit Generation for Mobile Devices. *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, Kralendijk, Bonaire, 2012.
9. Huang Y, Chapman P, Evans D. Privacy-Preserving Applications on Smartphones. *Proceedings of the USENIX Workshop on Hot Topics in Security*, San Francisco, CA, USA, 2011.
10. Freedman M, Nissim K, Pinkas B. Efficient private matching and set intersection. *EUROCRYPT*, Interlaken, Switzerland, 2004.
11. Hirt M, Sako K. Efficient Reciept-free voting based on homomorphic encryption. *Proceedings of the International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT)*, Bruges, Belgium, 2000.
12. Zhong G, Goldberg I, Hengartner U. Louis, Lester and Pierre: Three Protocols for Location Privacy. *Privacy Enhancing Technologies Symposium*, Ottawa, Canada, 2007.
13. De Cristofaro E, Tsudik G. Practical private set intersection protocols with linear complexity. *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, Canary Islands, Spain, 2010.
14. Camenisch J, Zaverucha G. Private intersection of certified sets. *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, Christ Church, Barbados, 2009.
15. Kim M, Lee HT, Cheon JH. Mutual private set intersection with linear complexity. *Proceedings of the international conference on Information Security Applications*, Jeju Island, Korea, 2011.
16. De Cristofaro E, Kim J, Tsudik G. Linear-complexity private set intersection protocols secure in malicious model. *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Singapore, 2010.
17. Kissner L, Song D. Privacy-preserving set operations. *Advances in Cryptology (CRYPTO)*, Santa Barbara, CA, USA, 2005.
18. Kreuter B, Shelat A, Shen Ch. Billion-Gate Secure Computation with Malicious Adversaries. *Proceedings of the USENIX Security Symposium*, Bellevue, WA, USA, 2012.
19. Huang Y, Evans D, Katz J. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? *Proceedings of the isoc Network and Distributed Systems Security (NDSS) Symposium*, San Diego, CA, USA, 2012.
20. Carter H, Mood B, Traynor P, Butler K. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. *Proceedings of the USENIX Security Symposium*, Washington, D.C., USA, 2013.
21. Nipane N, Dacosta I, Traynor P. "Mix-In-Place" anonymous networking using secure function evaluation. *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Orlando, FL, USA, 2011.
22. Kerschbaum F, Schropfer A, Dahlmeier D, Biswas D. On the Practical Importance of Communication Complexity for Secure Multi-Party Computation Protocols. *Proceedings of the ACM Symposium on Applied Computing (SAC)*, Honolulu, HI, USA, 2009.
23. Nakamura T, Inenaga S, Ikeda D, Baba K, Yasuura H. Anonymous Authentication Systems Based on Private Information Retrieval. *International Comference on Networked Digital Technologies (NDT)*, Ostrava, The Czech Republic, 2009.
24. Bethencourt J, Song D, Waters B. Analysis-Resistant Malware. *Proceedings of the ISOC Network and Distributed Systems Security (NDSS) Symposium*, 2008.
25. Ostrovsky R, III WES. Private Searching On Streaming Data. *Journal of Cryptology* 2007; **20**(4):397–430.
26. Ramachandran A, Zhou Z, Huang D. Computing Cryptographic Algorithms in Portable and Embedded Devices. *Proceedings of the IEEE International Conference on Portable Information Devices (PORTABLE)*, Orlando, FL, USA, 2007.
27. Goldreich O. *Foundations of Cryptography Volume II: Basic Applications*. Cambridge University Press, 1996.
28. Odelu V, Das A, Goswami A. An effective and secure key-management scheme for hierarchical access control in e-medicine system. *Journal of Medical Systems* 2013; **37**(2):1–18.
29. Katz J, Lindell Y. *Introduction to Modern Cryptography*. Chapman and Hall/CRC, 2007.
30. Lindell Y, Pinkas B. A Proof of Yao's Protocol for Secure Two-Party Computation. *Journal of Cryptology* 2009; **22**(2):161–188.
31. Carter H, Amrutkar C, Dacosta I, Traynor P. Efficient Oblivious Computation Techniques for Privacy-Preserving Mobile Applications. *Technical Report GT-CS-11-11*, College of Computing, Georgia Institute of Technology 2011.
32. Kustans E. Shark for root. https://play.google.com/store/apps/details?id=lv.n3o.shark 2012.
33. The Wireshark Foundation. Wireshark. https://www.wireshark.org/ 2012.
34. Google. Android project. http://source.android.com 2010.
35. Naor M, Pinkas B. Efficient oblivious transfer protocols. *Proceedings of the ACM-SIAM symposium on Discrete algorithms*, Washington, D.C., USA, 2001.

Reuse It Or Lose It: More Efficient Secure Computation Through Reuse of Encrypted Values

Benjamin Mood
University of Florida
bmood@ufl.edu

Debayan Gupta
Yale University
debayan.gupta@yale.edu

Kevin R. B. Butler
University of Florida
butler@ufl.edu

Joan Feigenbaum
Yale University
joan.feigenbaum@yale.edu

ABSTRACT

Two-party secure-function evaluation (SFE) has become significantly more feasible, even on resource-constrained devices, because of advances in server-aided computation systems. However, there are still bottlenecks, particularly in the input-validation stage of a computation. Moreover, SFE research has not yet devoted sufficient attention to the important problem of retaining state after a computation has been performed so that expensive processing does not have to be repeated if a similar computation is done again. This paper presents PartialGC, an SFE system that allows the reuse of encrypted values generated during a garbled-circuit computation. We show that using PartialGC can reduce computation time by as much as 96% and bandwidth by as much as 98% in comparison with previous outsourcing schemes for secure computation. We demonstrate the feasibility of our approach with two sets of experiments, one in which the garbled circuit is evaluated on a mobile device and one in which it is evaluated on a server. We also use PartialGC to build a privacy-preserving "friend-finder" application for Android. The reuse of previous inputs to allow stateful evaluation represents a new way of looking at SFE and further reduces computational barriers.

1. INTRODUCTION

Secure function evaluation, or *SFE*, allows multiple parties to jointly compute a function while maintaining input and output privacy. The two-party variant, known as 2P-SFE, was first introduced by Yao in the 1980s [39] and was largely a theoretical curiosity. Developments in recent years have made 2P-SFE vastly more efficient [18, 27, 38]. However, computing a function using SFE is still usually much slower than doing so in a non-privacy-preserving manner.

As mobile devices become more powerful and ubiquitous, users expect more services to be accessible through them. When SFE is performed on mobile devices (where resource constraints are tight), it is extremely slow – *if* the computation can be run at all without exhausting the memory, which can happen for non-trivial input sizes and algorithms [8]. One way to allow mobile devices to perform SFE is to use a server-aided computational model [8, 22], allowing the majority of an SFE computation to be "outsourced" to a more powerful device while still preserving privacy. Past approaches, however, have not considered the ways in which mobile computation differs from the desktop. Often, the mobile device is called upon to perform *incremental* operations that are continuations of a previous computation.

Consider, for example, a "friend-finder" application where the location of users is updated periodically to determine whether a contact is in proximity. Traditional applications disclose location information to a central server. A privacy-preserving friend-finder could perform these operations in a mutually oblivious fashion. However, every incremental location update would require a full re-evaluation of the function with fresh inputs in a standard SFE solution. Our examination of an outsourced SFE scheme for mobile devices by Carter et al. [8] (hereon CMTB), determined that the cryptographic consistency checks performed on the inputs to an SFE computation *themselves* can constitute the greatest bottleneck to performance.

Additionally, many other applications require the ability to save state, a feature that current garbled-circuit implementations do not possess. The ability to save state and reuse an intermediate value from one garbled circuit execution in another would be useful in many other ways, *e.g.*, we could split a large computation into a number of smaller pieces. Combined with efficient input validation, this becomes an extremely attractive proposition.

In this paper, we show that it is possible to reuse an encrypted value in an outsourced SFE computation (we use a cut-and-choose garbled circuit protocol) even if one is restricted to primitives that are part of standard garbled circuits. Our system, PartialGC, which is based on CMTB, provides a way to take encrypted output wire values from one SFE computation, save them, and then reuse them as input wires in a new garbled circuit. Our method vastly reduces the number of cryptographic operations compared to the trivial mechanism of simply XOR'ing the results with a one-time pad, which requires either generating inside the circuit, or inputting, a very large one-time pad, both complex operations. Through the use of improved input validation mechanisms proposed by shelat and Shen [38] (hereon sS13) and new methods of *partial input* gate checks and evaluation, we improve on previous proposals. There are other

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660285.

approaches to the creation of reusable garbled circuits [13, 10, 5], and previous work on reusing encrypted values in the ORAM model [30, 11, 31], but these earlier schemes have not been implemented. By contrast, we have implemented our scheme and found it to be both practical and efficient; we provide a performance analysis and a sample application to illustrate its feasibility (Section 6), as well as a simplified example execution (Appendix C).

By breaking a large program into smaller pieces, our system allows interactive I/O throughout the garbled circuit computation. To the best of our knowledge this is the first practical protocol for performing interactive I/O in the middle of a cut-and-choose garbled circuit computation.

Our system comprises three parties - a generator, an evaluator, and a third party ("the cloud"), to which the evaluator outsources its part of the computation. Our protocol is secure against a malicious adversary, assuming that there is no collusion with the cloud. We also provide a semi-honest version of the protocol.

Figure 1 shows how PartialGC works at a high level: First, a standard SFE execution (blue) takes place, at the end of which we "save" some intermediate output values. All further executions use intermediate values from previous executions. In order to reuse these values, information from both parties – the generator and the evaluator – has to be saved. In our protocol, it is the cloud – rather than the evaluator – that saves information. This allows multiple distinct evaluators to participate in a large computation over time by saving state in the cloud between different garbled circuit executions. For example, in a scenario where a mobile phone is outsourcing computation to a cloud, PartialGC can save the encrypted intermediate outputs to the cloud instead of the phone (Figure 2). This allows the phones to communicate with each other by storing encrypted intermediate values in the cloud, which is more efficient than requiring them to directly participate in the saving of values, as required by earlier 2P-SFE systems. Our friend finder application, built for an Android device, reflects this usage model and allows multiple friends to share their intermediate values in a cloud. Other friends use these saved values to check whether or not someone is in the same map cell as themselves without having to copy and send data.

By incorporating our optimizations, we give the following contributions:

1. *Reusable Encrypted Values* – We show how to reuse an encrypted value, using only garbled circuits, by mapping one garbled value into another.
2. *Reduced Runtime and Bandwidth* – We show how reusable encrypted values can be used in practice to reduce the execution time for a garbled-circuit computation; we get a 96% reduction in runtime and a 98% reduction in bandwidth over CMTB.
3. *Outsourcing Stateful Applications* – We show how our system increases the scope of SFE applications by allowing multiple evaluating parties over a period of time to operate on the saved state of an SFE computation without the need for these parties to know about each other.

The remainder of our paper is organized as follows: Section 2 provides some background on SFE. Section 3 introduces the concept of partial garbled circuits in detail. The PartialGC protocol and its implementation are described in Section 4, while its security is analyzed in Section 5. Section 6 evaluates PartialGC and introduces the friend finder application.

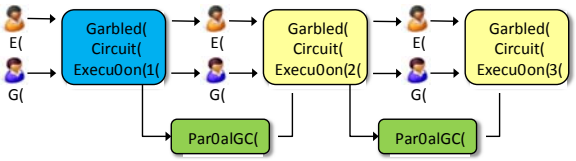


Figure 1: PartialGC Overview. E is evaluator and G is generator. The blue box is a standard execution that produces partial outputs (garbled values); yellow boxes represent executions that take partial inputs and produce partial outputs.

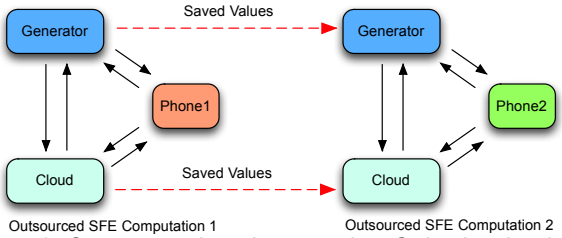


Figure 2: Our system has three parties. Only the cloud and generator have to save intermediate values - this means that we can have different phones in different computations.

Section 7 discusses related work and Section 8 concludes.

2. BACKGROUND

Secure function evaluation (SFE) addresses scenarios where two or more mutually distrustful parties $P_1, \ldots, P_n$, with private inputs $x_1, \ldots, x_n$, want to compute a given function $y_i = f(x_1, \ldots, x_n)$ (y_i is the output received by P_i), such that no P_i learns anything about any x_j or y_j, $i \neq j$ that is not logically implied by x_i and y_i. Moreover, there exists no trusted third party – if there was, the P_is could simply send their inputs to the trusted party, which would evaluate the function and return the y_is.

SFE was first proposed in the 1980s in Yao's seminal paper [39]. The area has been studied extensively by the cryptography community, leading to the creation of the first general purpose platform for SFE, Fairplay [32] in the early 2000s. Today, there exist many such platforms [6, 9, 16, 17, 26, 37, 40].

The classic platforms for 2P-SFE, including Fairplay, use garbled circuits. A garbled circuit is a Boolean circuit which is encrypted in such a way that it can be evaluated when the proper input wires are entered. The party that evaluates this circuit does not learn anything about what any particular wire represents. In 2P-SFE, the two parties are: the *generator*, which creates the garbled circuit, and the *evaluator*, which evaluates the garbled circuit. Additional cryptographic techniques are used for input and output; we discuss these later.

A two-input Boolean gate has four truth table entries. A two-input garbled gate also has a truth table with four entries representing 1s and 0s, but these entries are encrypted and can only be retrieved when the proper keys are used. The values that represent the 1s and 0s are random strings of bits. The truth table entries are permuted such that the evaluator cannot determine which entry she is able to decrypt, only that she is able to decrypt an entry. The entirety of a garbled gate is the four encrypted output values.

Each garbled gate is then encrypted in the following way: Each entry in the truth table is encrypted under the two input wires, which leads to the result, $truth_i = Enc(input_x || input_y) \oplus output_i$, where $truth_i$ is a value in the truth table,

$input_x$ is the value of input wire x, $input_y$ is the value of input wire y, and $output_i$ is the non-encrypted value, which represents either 0 or 1.We use AES as the Enc function. If the evaluator has $input_x$ and $input_y$, then she can also receive $output_i$, and the encrypted truth tables are sent to her for evaluation.

For the evaluator's input, 1-out-of-2 oblivious transfers (OTs) [1, 20, 34, 35] are used. In a 1-out-of-2 OT, one party offers up two possible values while the other party selects one of the two values without learning the other. The party that offers up the two values does not learn which value was selected. Using this technique, the evaluator gets the wire labels for her input without leaking information.

The only way for the evaluator to get a correct output value from a garbled gate is to know the correct decryption keys for a specific entry in the truth table, as well as the location of the value she has to decrypt.

During the permutation stage, rather than simply randomly permuting the values, the generator permutes values based on a specific bit in $input_x$ and $input_y$, such that, given $input_x$ and $input_y$ the evaluator knows that the location of the entry to decrypt is $bit_x * 2 + bit_y$. These bits are called the *permutation bits*, as they show the evaluator which entry to select based on the permutation; this optimization, which does not leak any information, is known as *point and permute* [32].

2.1 Threat Models

Traditionally, there are two threat models discussed in SFE work, semi-honest and malicious. The above description of garbled circuits is the same in both threat models. In the semi-honest model users stay true to the protocol but may attempt to learn extra information from the system by looking at any message that is sent or received. In the malicious model, users may attempt to change anything with the goal of learning extra information or giving incorrect results without being detected; extra techniques must be added to achieve security against a malicious adversary.

There are several well-known attacks a malicious adversary could use against a garbled circuit protocol. A protocol secure against malicious adversaries must have solutions to all potential pitfalls, described in turn:

Generation of incorrect circuits If the generator does not create a correct garbled circuit, he could learn extra information by modifying truth table values to output the evaluator's input; he is limited only by the external structure of the garbled circuit the evaluator expects.

Selective failure of input If the generator does not offer up correct input wires to the evaluator, and the evaluator selects the wire that was not created properly, the generator can learn up to a single bit of information based on whether the computation produced correct outputs.

Input consistency If either party's input is not consistent across all circuits, then it might be possible for extra information to be retrieved.

Output consistency In the two-party case, the output consistency check verifies that the evaluator did not modify the generator's output before sending it.

2.1.1 Non-collusion

CMTB assumes non-collusion, as quoted below:
"The outsourced two-party SFE protocol securely computes a function f(a,b) in the following two corruption scenarios: (1)The cloud is malicious and non-cooperative with respect to the rest of the parties, while all other parties are semi-honest, (2)All but one party is malicious, while the cloud is semi-honest."

This is the standard definition of non-collusion used in server-aided works such as Kamara et al. [22]. Non-collusion does not mean the parties are trusted; it only means the two parties are not working together (i.e. both malicious). In CMTB, any individual party that attempts to cheat to gain additional information will still be caught, but collusion between multiple parties could leak information. For instance, the generator could send the cloud the keys to decrypt the circuit and see what the intermediate values are of the garbled function.

3. PARTIAL GARBLED CIRCUITS

We introduce the concept of *partial garbled circuits* (PGCs), which allows the encrypted wire outputs from one SFE computation to be used as inputs to another. This can be accomplished by *mapping* the encrypted output wire values to valid input wire values in the next computation. In order to better demonstrate their structure and use, we first present PGCs in a semi-honest setting, before showing how they can aid us against malicious adversaries.

3.1 PGCs in the Semi-Honest Model

In the semi-honest model, for each wire value, the generator can simply send two values to the evaluator, which transforms the wire label the evaluator owns to work in another garbled circuit. Depending on the point and permute bit of the wire label received by the evaluator, she can map the value from a previous garbled circuit computation to a valid wire label in the next computation.

Specifically, for a given wire pair, the generator has wires w_0^{t-1} and w_1^{t-1}, and creates wires w_0^t and w_1^t. Here, t refers to a particular computation in a series, while 0 and 1 correspond to the values of the point and permute bits of the $t-1$ values. The generator sends the values $w_0^{t-1} \oplus w_0^t$ and $w_1^{t-1} \oplus w_1^t$ to the evaluator. Depending on the point and permute bit of the w_i^{t-1} value she possesses, the evaluator selects the correct value and then XORs her w_i^{t-1} with the $(w_i^{t-1} \oplus w_i^t)$ value, thereby giving her w_i^t, the valid partial input wire.

3.2 PGCs in the Malicious Model

In the malicious model we must allow the evaluation of a circuit with partial inputs and verification of the mappings, while preventing a selective failure attack. The following features are necessary to accomplish these goals:

1. Verifiable Mapping

The generator G is able to create a secure mapping from a saved garbled wire value into a new computation that can be checked by the evaluator E, without E being able to reverse the mapping. During the evaluation and check phase, E must be able to verify the mapping G sent. G must have either committed to the mappings before deciding the partition of evaluation and check circuits, or never learned which circuits are in the check versus the evaluation sets.

2. Partial Generation and Partial Evaluation

G creates the garbled gates necessary for E to enter the previously output intermediate encrypted values into the next garbled circuit. These garbled gates are called *partial input gates*. As shown in Figure 3 each garbled circuit is

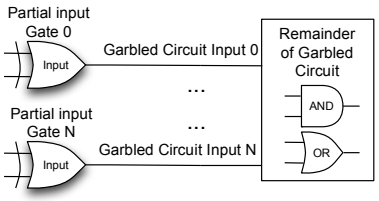


Figure 3: This figure shows how we create a single *partial input gate* for each input bit for each circuit and then link the *partial input gates* to the remainder of the circuit.

made up of two pieces: the partial input gates and the remainder of the garbled circuit.

3. Revealing Incorrect Transformations

Our last goal is to let E inform G that incorrect values have been detected. Without a way to limit leakage, G could gain information based on whether or not E informs G that she caught him cheating. This is a selective failure attack and is not present in our protocol.

4. PARTIAL GC PROTOCOL

We start with the CMTB protocol and add cut-and-choose operations from sS13 before introducing the mechanisms needed to save and reuse values. We defer to the original papers for full details of the outsourced oblivious transfer [8] and the generator's input consistency check [38] subprotocols that we use as primitives in our protocol.

Our system operates in the same threat model as CMTB (see Section 2.1.1): we are secure against a malicious adversary under the assumption of non-collusion. A description of the CMTB protocol is available in Appendix A.

4.1 Preliminaries

There are three participants in the protocol:

Generator – The generator is the party that generates the garbled circuit for the 2P-SFE.

Evaluator –The evaluator is the other party in the 2P-SFE, which is outsourcing computation to a third party, the cloud.

Cloud – The cloud is the party that executes the garbled circuit outsourced by the evaluator.

Notation

C_i - The ith circuit.

$CKey_i$ - Circuit key used for the free XOR optimization [25]. The key is randomly generated and then used as the difference between the 0 and 1 wire labels for a circuit C_i.

$CSeed_i$ - This value is created by the generator's PRNG and is used to generate a particular circuit C_i.

$POut\#_{i,j}$ - The *partial output* values are the encrypted wire values output from an SFE computation. These are encrypted garbled circuit values that can be reused in another garbled circuit computation. # is replaced in our protocol description with either a 0, 1, or x, signifying whether it represents a 0, 1, or an unknown value (from the cloud's point of view). i denotes the circuit the $POut$ value came from and j denotes the wire of the $POut_i$ circuit.

$PIn\#_{i,j}$ - The *partial input* values are the re-entered $POut$ values after they have been obfuscated to remove the circuit key from the previous computation. These values are input to the *partial input gates*. #, i, and j, are the same as above.

$GIn\#_{i,j}$ - The *garbled circuit input* values are the results of the partial input gates and are input into the remaining garbled circuit, as shown in Figure 3. #, i, and j, are the same as above.

Partial Input Gates - These are garbled gates that take in PIn values and output GIn values. Their purpose is to transform the PIn values into values that are under $CKey_i$ for the current circuit.

4.2 Protocol

Each computation is self-contained; other than what is explicitly described as saved in the protocol, each value or property is only used for a single part of the computation (*i.e.* randomness is different across computations).

Algorithm 0: PartialComputation

```
Input  : Circuit_File, Bit_Security, Number_of_Circuits, Inputs,
         Is_First_Execution
Output: Circuit File Output
Cut_and_Choose(is_First_Execution)
Eval_Garbled_Input      Evaluator_Input(Eval_Select_Bits,
Possible_Eval_Input)
Generator_Input_Check(Gen_Input)
Partial_Garbled_Input   Partial_Input(Partial_Output_time-1)
Garbled_Output, Partial_Output
Circuit_Execution(Garbled_Input (Gen, Eval, Partial))
Circuit_Output(Garbled_Output)
Partial_Output(Partial_Output)
```

Common Inputs: The program circuit file, the bit level security, the circuit level security (number of circuits) S, and encryption and commitment functions.

Private Inputs: The evaluator's input *evlInput* and generator's input *genInput*.

Outputs: The evaluator and generator can both receive garbled circuit outputs.

Phase 1: Cut-and-choose

We modify the cut-and-choose mechanism described in sS13 as we have an extra party involved in the computation. In this cut-and-choose, the cloud selects which circuits are evaluation circuits and which circuits are check circuits,

$$circuitSelection = rand()$$

where *circuitSelection* is a bit vector of size S; N evaluation circuits and $S-N$ check circuits are selected where $N = \frac{2}{5}S$. The generator does not learn the circuit selection.

The generator generates garbled versions of his input and circuit seeds for each circuit. He encrypts these values using unique 1-time XOR pad keys. For $0 \leq i < S$,

$$CSeed_i = rand()$$
$$garbledGenInput_i = garble(genInput, rand())$$
$$checkKey_i = rand()$$
$$evlKey_i = rand()$$
$$encSeedIn_i = CSeed_i \oplus evlKey_i$$
$$encGarbledIn_i = garbledGenInput_i \oplus checkKey_i$$

The cloud and generator perform an oblivious transfer where the generator offers up decryption keys for his input and decryption keys for the circuit seed for each circuit. The cloud can select the key to decrypt the generator's input or the key to decrypt the circuit seed for a circuit but not both. For each circuit, if the cloud selects the decryption key for the circuit seed in the oblivious transfer, then the circuit is used as a check circuit.

Algorithm 1: Cut_and_Choose

```
Input  : is_First_Execution
if is_First_Execution then
    circuitSelection ← rand() // bit-vector of size S
N ← (2/5)S // Number of evaluation circuits
//Generator creates his garbled input and circuit seeds for each
circuit
for i ← 0 to S do
    CSeed_i ← rand()
    garbledGenInput_i ← garble(genInput, rand())
    //generator creates or loads keys
    if is_First_Execution then
        checkKey_i ← rand()
        evlKey_i ← rand()
    else
        loadKeys();
        checkKey_i ← hash(loadedCheckKey_i)
        evlKey_i ← hash(loadedEvlKey_i)
    // encrypts using unique 1-time XOR pads
    encSeedIn_i ← CSeed_i ⊕ evlKey_i
    encGarbledIn_i ← garbledGenInput_i ⊕ checkKey_i
if is_First_Execution then
    // generator offers input OR keys for each circuit seed
    selectedKeys ←
    OT(circuitSelection, {evlKey, checkKey})
else
    loadSelectedKeys()
for i ← 0 to S do
    genSendToEval(hash(checkKey_i),
    hash(evaluationKey_i))
for i ← 0 to S do
    cloudSendToEval(hash(selectedKey_i), isCheckCircuit_i)
// If all values match, the evaluator learns split, else abort.
for i ← 0 to S do
    j ← isCheckCircuit_i
    correct ← (recievedGen_{i,j} == recievedEvl_i)
    if !correct then
        abort()
```

$$selectedKeys = OT(circuitSelection, \{evlKey, checkKey\})$$

If the cloud selects the key for the generator's input then a given circuit is used as an evaluation circuit. Otherwise, the key for the circuit seed was selected and the circuit is a check circuit. The decryption keys are saved by both the generator and cloud in the event a computation uses saved values from this computation.

The generator sends the encrypted garbled inputs and check circuit information for all circuits to the cloud. The cloud decrypts the information he can decrypt using its keys.

The evaluator must also learn the circuit split. The generator sends a hash of each possible encryption key the cloud could have selected to the evaluator for each circuit as an ordered pair. For $0 \leq i < S$,

$$genSend(hash(checkKey_i), hash(evaluationKey_i))$$

The cloud sends a hash of the value received to the evaluator for each circuit. The cloud also sends bits to indicate which circuits were selected as check and evaluation circuits to the evaluator. For $0 \leq i < S$,

$$cloudSend(hash(selectedKey_i), isCheckCircuit_i)$$

The evaluator compares each hash the cloud sent to one of the hashes the generator sent, which is selected by the circuit selection sent by the cloud. For $0 \leq i < S$,

$$j = isCheckCircuit_i$$
$$correct = (receivedGen_{i,j} == receivedEvl_i)$$

If all values match, the evaluator uses the $isCheckCircuit_i$ to learn the split between check and evaluator circuits. Otherwise the evaluator safely aborts.

We only perform the cut-and-choose oblivious transfer for the initial computation. For any subsequent computations, the generator and evaluator hash the saved decryption keys and use those hashes as the new encryption and decryption keys. The circuit split selected by the cloud is saved and stays the same across computations.

Phase 2: Oblivious Transfer

Algorithm 2: Evaluator_Input

```
Input  : Eval_Select_Bits, Possible_Eval_Input
Output: Eval_Garbled_Input
// cloud gets selected input wires // generator offers both
possible input wire values for each input wire; evaluator selects
its input
outSeeds = BaseOOT(bitsEvl, possibleInputs).
// the generator sends unique IKey values for each circuit to the
evaluator
for i ← 0 to S do
    genSendToEval(IKey_i)
// the evaluator sends IKey values for all evaluation circuits to
the cloud
for i ← 0 to S do
    if !isCheckCircuit(i) then
        EvalSendToCloud(IKey_i)
// cloud uses this to learn appropriate inputs
for i ← 0 to S do
    for j ← 0 to len(evlInputs) do
        if !isCheckCircuit(i) then
            inputEvl_ij ← hash(IKeys_i, outSeeds_j)
return inputEvl
```

We use the base outsourced oblivious transfer (OOT) of CMTB. In this transfer the generator inputs both possible input wire values for each evaluator's input wire while the evaluator inputs its own input. After the OOT is performed, the cloud has the selected input wire values, which represent the evaluator's input.

As with CMTB, which uses the results from a single OOT as seeds to create the evaluator's input for all circuits, the cloud in our system also uses seeds from a single base OT (called "BaseOOT" below) to generate the input for the evaluation circuits. The cloud receives the seeds for each input bit selected by the evaluator.

$$outSeeds = BaseOOT(evlInputSeeds, evlInput).$$

The generator creates unique keys, $IKey$, for each circuit and sends each key to the evaluator. The evaluator sends the keys for the evaluation circuits to the cloud. The cloud then uses these values to attain the evaluator's input. For $0 \leq i < S$, for $0 \leq j < len(evlInputs)$ where $!isCheckCircuit(i)$,

$$inputEvl_{ij} = hash(IKey_i, outSeeds_j)$$

Phase 3: Generator's Input Consistency Check

We use the input consistency check of sS13. In this check, a universal hash is used to prove consistency of the generator's input across each evaluation circuit. Simply put, if the hash is different in any of the evaluation circuits, we know the generator did not enter consistent input. More formally, a hash of the generator's input is taken for each circuit. For $0 < i < S$ where $!isCheckCircuit(i)$,

$$t_i = UHF(garbledGenInput_i, C_i)$$

Algorithm 3: Generator_Input_Check

```
Input  : Generator_Input
// The cloud takes a hash of the generator's input or each
evaluation circuit for i ← 0 to S do
    if isCheckCircuit(i) then
        t_i ← UHF(garbledGenInput_i)
//If a single hash is different then the cloud knows the generator
tried to cheat.
correct ← ((t_0 == t_1)&(t_0 == t_2)& ... &(t_0 == t_{N-1}))
if !correct then
    abort()
```

The results of these universal hashes are compared. If a single hash is different then the cloud knows the generator tried to cheat and safely aborts.

$$correct = ((t_0 == t_1)\&(t_0 == t_2)\& \ldots \&(t_0 == t_{N-1}))$$

Phase 4: Partial Input Gate Generation, Check, and Evaluation

Generation

For $0 \leq i < S$, for $0 \leq j < len(savedWires)$, the generator creates a *partial input gate*, which transforms a wire's saved values, $POut0_{i,j}$ and $POut1_{i,j}$, into wire values that can be used in the current garbled circuit execution, $GIn0_{i,j}$ and $GIn1_{i,j}$. For each circuit, C_i, the generator creates a pseudorandom transformation value R_i, to assist with the transformation.

For each set of $POut0_{i,j}$ and $POut1_{i,j}$, the generator XORs each value with R_i. Both results are then hashed, and put through a function to determine the new permutation bit, as hashing removes the old permutation bit.

$$t0 = hash(POut0_{i,j} \oplus R_i)$$
$$t1 = hash(POut1_{i,j} \oplus R_i)$$
$$PIn0_{i,j}, PIn1_{i,j} = setPPBitGen(t0, t1)$$

This function, *setPPBitGen*, pseudo-randomly finds a bit that is different between the two values of the wire and notes that bit to be the permutation bit. *setPPBitGen* is seeded from $CSeed_i$, allowing the cloud to regenerate these values for the check circuits.

For each $PIn0_{i,j}, PIn1_{i,j}$ pair, a set of values, $GIn0_{i,j}$ and $GIn1_{i,j}$, are created under the master key of C_i, $CKey_i$, – where $CKey_i$ is the difference between 0 and 1 wire labels for the circuit. In classic garbled gate style, two truth table values, $TT0_{i,j}$ and $TT1_{i,j}$, are created such that:

$$TT0_{i,j} \oplus PIn0_{i,j} = GIn0_{i,j}$$
$$TT1_{i,j} \oplus PIn1_{i,j} = GIn1_{i,j}$$

The truth table, $TT0_{i,j}$ and $TT1_{i,j}$, is permuted so that the permutation bits of $PIn0_{i,j}$ and $PIn1_{i,j}$ tell the cloud which entry to select. Each *partial input gate*, consisting of the permuted $TT0_{i,j}$, $TT1_{i,j}$ values and the bit location from *setPPBitGen* is sent to the cloud. Each R_i is also sent to the cloud.

Check

For $0 \leq i < S$ where $isCheckCircuit(i)$, for $0 \leq j < len(savedWires)$, the cloud receives the truth table information, $TT0_{i,j}, TT1_{i,j}$, and bit location from *setPPBitGen*, and proceeds to regenerate the gates based on the check circuit information. The cloud uses R_i (sent by the generator), $POut0_{i,j}$ and $POut1_{i,j}$ (saved during the previous execution), and $CSeed_i$ (recovered during the cut-and-choose) to generate the *partial input gates* in the same manner as described previously. The cloud then compares these gates to those the generator sent. If any gate does not match, the cloud knows the generator tried to cheat and safely aborts.

Algorithm 4: Partial_Input

```
Input  : Partial_Output
Output: Partial_Garbled_Input
// Generation: the generator creates a partial input gate, which
transforms a wire's saved values, POut0_{i,j} and POut1_{i,j}, into
values that can be used in the current garbled circuit execution,
GIn0_{i,j} and GIn1_{i,j}.
for i ← 0 to S do
    R_i ← PRNG.random()
    for j ← 0 to len(savedWires) do
        t0 ← hash(POut0_{i,j} ⊕ R_i)
        t1 ← hash(POut1_{i,j} ⊕ R_i)
        PIn0_{i,j}, PIn1_{i,j} ← setPPBitGen(t0, t1)
        GIn0_{i,j} ← TT0_{i,j} ⊕ PIn0_{i,j}
        GIn1_{i,j} ← TT1_{i,j} ⊕ PIn1_{i,j}
        GenSendToCloud( Permute([TT0_{i,j}, TT1_{i,j}]),
        permute_bit_locations )
    GenSendToCloud(R_i)
// Check: The cloud checks the gates to make sure the generator
didn't cheat
for i ← 0 to S do
    if isCheckCircuit(i) then
        for j ← 0 to len(savedWires) do
            // the cloud has received the truth table
            information, TT0_{i,j}, TT1_{i,j}, bit locations from
            setPPBitGen, and R_i
            correct ← (generateGateFromInfo() ==
            receivedGateFromGen())
            // If any gate does not match, the cloud knows the
            generator tried to cheat.
            if !correct then
                abort();
// Evaluation
for i ← 0 to S do
    if !isCheckCircuit(i) then
        for j ← 0 to len(savedWires) do
            //The cloud, using the previously saved POutx_{i,j}
            value, and the location (point and permute) bit sent
            by the generator, creates PInx_{i,j}
            PInx_{i,j} ←
            setPPBitEval(hash(R_i ⊕ POutx_{i,j}), location)
            // Using PInx_{i,j}, the cloud selects the proper
            truth table entry TTx_{i,j} from either TT0_{i,j} or
            TT1_{i,j} to decrypt
            // Creates GInx_{i,j} to enter into the garbled circuit
            GInx_{i,j} ← TTx_{i,j} ⊕ POutx_{i,j}
return GIn;
```

Evaluation

For $0 \leq i < S$ where $!isCheckCircuit(i)$, for $0 \leq j < len(savedWires)$ the cloud receives the truth table information, $TTa_{i,j}, TTb_{i,j}$ and bit location from *setPPBitGen*. a and b are used to denote the two permuted truth table values. The cloud, using the previously saved $POutx_{i,j}$ value, creates the $PInx_{i,j}$ value:

$$PInx_{i,j} = setPPBitEval(hash(R_i \oplus POutx_{i,j}), location)$$

location is the location of the point and permute bit sent by the generator. Using the point and permute bit of $PInx_{i,j}$, the cloud selects the proper truth table entry $TTx_{i,j}$ from either $TTa_{i,j}$ or $TTb_{i,j}$ to decrypt, creates $GInx_{i,j}$ and then enters $GInx_{i,j}$ into the garbled circuit.

$$GInx_{i,j} = TTx_{i,j} \oplus POutx_{i,j}$$

Phase 5: Circuit Generation and Evaluation

Algorithm 5: Circuit_Execution

```
Input  : Generator_Input, Evaluator_Input, Partial_Input
Output: Partial_Output, Garbled_Output
// The generator generates each garbled gate and sends it to the
cloud. Depending on whether the circuit is a check or evaluation
circuit, the cloud verifies that the gate is correct or evaluates the
gate.
for i ← 0 to S do
    for j ← 0 to len(circuit) do
        g ← genGate(C_i, j)
        send(g)
// the cloud receives all gates for all circuits, and then checks
OR evaluates each circuit
for i ← 0 to S do
    for j ← 0 to len(circuit) do
        g ← recvGate()
        if isCheckCircuit(i) then
            if ! verifyCorrect(g) then
                abort()
        else
            eval(g)
return Partial_Output, Garbled_Output
```

Circuit Generation

The generator generates each garbled gate for each circuit and sends them to the cloud. Since the generator does not know the check and evaluation circuit split, nothing changes for the generation for check and evaluation circuits. For $0 \leq i < S$, For $0 \leq j < len(circuit)$,

$$g = genGate(C_i, j), send(g)$$

Circuit Evaluation and Check

The cloud receives each garbled gate for all circuits. For evaluation circuits the cloud evaluates those garbled gates. For check circuits the cloud generates the correct gate, based on the circuit seed, and is able to verify it is correct.

For $0 \leq i < S$, For $0 \leq j < len(circuit)$, $g = recvGate()$, $if(isCheckCircuit(j))$ $verifyCorrect(g)$ $else$ $eval(g)$

If a garbled gate is found not to be correct, the cloud informs the evaluator and generator of the incorrect gate and safely aborts.

Phase 6: Output and Output Consistency Check

Algorithm 6: Circuit_Output

```
Input  : Garbled_Output
// a MAC of the output is generated inside the garbled circuit,
and both the resulting garbled circuit output and the MAC are
encrypted under a one-time pad.
outEvlComplete = outEvl||MAC(outEvl)
result = (outEvlMAC == MAC(outEvl))
if !result then
    abort() // output check fail
```

As the final step of the garbled circuit execution, a MAC of the output is generated inside the garbled circuit, based on a k-bit secret key entered into the function.

$$outEvlComplete = outEvl||MAC(outEvl)$$

Both the resulting garbled circuit output and the MAC are encrypted under a one-time pad. The generator can also have output verified in the same manner. The cloud sends the corresponding encrypted output to each party.

The generator and evaluator then decrypt the received ciphertext, perform a MAC over real output, and verify the cloud did not modify the output by comparing the generated MAC with the MAC calculated within the garbled circuit.

$$result = (outEvlMAC == MAC(outEvl))$$

Phase 7: Partial Output

Algorithm 7: Partial_Output

```
Input  : Partial_Output
for i ← 0 to S do
    for j ← 0 to len(Partial_Output) do
        //The generator saves both possible wire values
        GenSave(Partial_Output0_{i,j})
        GenSave(Partial_Output1_{i,j})
for i ← 0 to S do
    for j ← 0 to len(Partial_Output) do
        if isCheckCircuit(i) then
            EvlSave(Partial_Output0_{i,j})
            EvlSave(Partial_Output1_{i,j})
        else
            // circuit is evaluation circuit
            EvlSave(Partial_OutputX_{i,j})
```

The generator saves both possible wire values for each partial output wire. For each evaluation circuit the cloud saves the partial output wire value. For check circuits the cloud saves both possible output values.

4.3 Implementation

As with most garbled circuit systems there are two stages to our implementation. The first stage is a compiler for creating garbled circuits, while the second stage is an execution system to evaluate the circuits.

We modified the KSS12 [27] compiler to allow for the saving of intermediate wire labels and loading wire labels from a different SFE computation. By using the KSS12 compiler, we have an added benefit of being able to compare circuits of almost identical size and functionality between our system and CMTB, whereas other protocols compare circuits of sometimes vastly different sizes.

For our execution system, we started with the CMTB system and modified it according to our protocol requirements. PartialGC automatically performs the output consistency check, and we implemented this check at the circuit level. We became aware and corrected issues with CMTB relating to too many primitive OT operations performed in the outsourced oblivious transfer when using a high circuit parameter and too low a general security parameter in general. The fixes reduced the run-time of the OOT.

5. SECURITY OF PARTIALGC

In this section, we provide a basic proof sketch of the PartialGC protocol, showing that our protocol preserves the standard security guarantees provided by traditional garbled circuits - that is, none of the parties learns anything about the private inputs of the other parties that is not logically implied by the output it receives. Since we borrow heavily from [8] and [38], we focus on our additions, and defer to the original papers for detailed proofs of those protocols. Due to space constraints, we do not provide a formal proof here; a complete proof will be provided in the technical report.

We know that the protocol described in [8] allows us to garble individual circuits and securely outsource their evaluation. In this paper, we modify certain portions of the protocol to allow us to transform the output wire values from a previous circuit execution into input wire values in a new

circuit execution. These transformed values, which can be checked by the evaluator, are created by the generator using circuit "seeds."

We also use some aspects of [38], notably their novel cut-and-choose technique which ensures that the generator does not learn which circuits are used for evaluation and which are used for checking - this means that the generator must create the correct transformation values for all of the cut-and-choose circuits.

Because we assume that the CMTB garbled circuit scheme can securely garble any circuit, we can use it individually on the circuit used in the first execution and on the circuits used in subsequent executions. We focus on the changes made at the end of the first execution and the beginning of subsequent executions which are introduced by PartialGC.

The only difference between the initial garbled circuit execution and any other garbled circuit in CMTB is that the output wires in an initial PartialGC circuit are stored by the cloud, and are not delivered to the generator or the evaluator. This prevents them from learning the output wire labels of the initial circuit, but cannot be less secure than CMTB, since no additional steps are taken here.

Subsequent circuits we wish to garble differ from ordinary CMTB garbled circuits only by the addition, before the first row of gates, of a set of partial input gates. These gates don't change the output along a wire, but differ from normal garbled gates in that the two possible labels for each input wire are not chosen randomly by the generator, but are derived by using the two labels along each output wire of the initial garbled circuit.

This does not reduce security. In PartialGC, the input labels for partial input gates have the same property as the labels for ordinary garbled input gates: the generator knows both labels, but does not know which one corresponds to the evaluator's input, and the evaluator knows only the label corresponding to its input, but not the other label. This is because the evaluator's input is exactly the output of the initial garbled circuit, the output labels of which were saved by the evaluator. The evaluator does not learn the other output label for any of the output gates because the output of each garbled gate is encrypted. If the evaluator could learn any output labels other than those which result from an evaluation of the garbled circuit, the original garbled circuit scheme itself would not be secure.

The generator, which also generated the initial garbled circuit, knows both possible input labels for all partial evaluation gates, because it has saved both potential output labels of the initial circuit's output gates. Because of the outsourced oblivious transfer used in CMTB, the generator did not know which input labels to use for the initial garbled circuit, and therefore will not have been able to determine the output labels for that circuit. Therefore, the generator will likewise not know which input labels are being used for subsequent garbled circuits.

Generator's Input Consistency Check

We use the generator's input consistency check from sS13. We note there is no problem with allowing the cloud to perform this check; for the generator's inconsistent input to pass the check, the cloud would have to see the malicious input and ignore it, which would violate the non-collusion assumption.

Correctness of Saved Values

Scenarios where either party enters incorrect values in the next computation reduce to previously solved problems in garbled circuits. If the generator does not use the correct values, then it reduces to the problem of creating an incorrect garbled circuit. If the evaluator does not use the correct saved values then it reduces to the problem of the evaluator entering garbage values into the garbled circuit execution; this would be caught by the output consistency check.

Abort on Check Failure

If any of the check circuits fail, the cloud reports the incorrect check circuit to both the generator and evaluator. At this point, the remaining computation and any saved values must be abandoned. However, as is standard in SFE, the cloud cannot abort on an incorrect evaluation circuit, even when she knows that it is incorrect.

Concatenation of Incorrect Circuits

If the generator produces a single incorrect circuit and the cloud does not abort, the generator learns that the circuit was used for evaluation, and not as a check circuit. This leaks no information about the input or output of the computation; to do that, the generator must corrupt a majority of the evaluation circuits without modifying a check circuit. An incorrect circuit that goes undetected in one execution has no effect on subsequent executions as long the total amount of incorrect circuits is less than the majority of evaluation circuits.

Using Multiple Evaluators

One of the benefits of our outsourcing scheme is that the state is saved at the generator and cloud allowing the use of different evaluators in each computation. Previously, it was shown a group of users working with a single server using 2P-SFE was not secure against malicious adversaries, as a malicious server and last k parties, also malicious, could replay their portion of the computation with different inputs and gain more information than they can with a single computation [15]. However, this is not a problem in our system as at least one of our servers, either the generator or cloud, must be semi-honest due to non-collusion, which obviates the attack stated above.

Threat Model

As we have many computations involving the same generator and cloud, we have to extend the threat model for how the parties can act in different computations. There can be no collusion in each singular computation. However, the malicious party can change between computations as long as there is no chain of malicious users that link the generator and cloud – this would break the non-collusion assumption.

6. PERFORMANCE EVALUATION

We now demonstrate the efficacy of PartialGC through a comparison with the CMTB outsourcing system. Apart from the cut-and-choose from sS13, PartialGC provides other benefits through generating partial input values after the first execution of a program. On subsequent executions, the partial inputs act to amortize overall costs of execution and bandwidth.

We demonstrate that the evaluator in the system can be a mobile device outsourcing computation to a more powerful system. We also show that other devices, such as server-class machines, can act as evaluators, to show the generality of this system. Our testing environment includes a 64-core server containing 1 TB of RAM, which we use to model both the Generator and Outsourcing Proxy parties. We run separate programs for the Generator and Outsourcing Proxy,

giving them each 32 threads. For the evaluator, we use a Samsung Galaxy Nexus phone with a 1.2 GHz dual-core ARM Cortex-A9 and 1 GB of RAM running Android 4.0, connected to the server through an 802.11 54 Mbps WiFi in an isolated environment. In our tests, which outsource the computation from a single server process we create that process on our 64-core server as well. We ran the CMTB implementation for comparison tests under the same setup.

6.1 Execution Time

The PartialGC system is particularly well suited to complex computations that require multiple stages and the saving of intermediate state. Previous garbled circuit execution systems have focused on single-transaction evaluations, such as computing the "millionaires" problem (i.e., a joint evaluation of which party inputs a greater value without revealing the values of the inputs) or evaluating an AES circuit.

Our evaluation considers two comparisons: the improvement of our system compared with CMTB without reusing saved values, and comparing our protocol for saving and reusing values against CMTB if such reuse was implemented in that protocol. We also benchmark the overhead for saving and loading values on a per-bit basis for 256 circuits, a necessary number to achieve a security parameter of 2^{-80} in the malicious model. In all cases, we run 10 iterations of each test and give timing results with 95% confidence intervals. Other than varying the number of circuits our system parameters are set for 80-bit security.

The programs used for our evaluation are exemplars of differing input sizes and differing circuit complexities:

Keyed Database: In this program, one party enters a database and keys to it while the other party enters a key that indexes into the database, receiving a database entry for that key. This is an example of a program expressed as a small circuit that has a very large amount of input.

Matrix Multiplication: Here, both parties enter 32-bit numbers to fill a matrix. Matrix multiplication is performed before the resulting matrix is output to both parties. This is an example of a program with a large amount of inputs with a large circuit.

Edit (Levenstein) Distance: This program finds the distance between two strings of the same length and returns the difference. This is an example of a program with a small number of inputs and a medium sized circuit.

Millionaires: In this classic SFE program, both parties enter a value, and the result is a one-bit output to each party to let them know whether their value is greater or smaller than that of the other party. This is an example of a small circuit with a large amount of input.

Gate counts for each of our programs can be found in Table 1. The only difference for the programs described above is the additional of a MAC function in PartialGC. We discuss the reason for this check in Section 6.4.

Table 2 shows the results from our experimental tests. In the best case, execution time was reduced by a factor of 32 over CMTB, from 1200 seconds to 38 seconds, a 96% speedup over CMTB. Ultimately, our results show that our system outperforms CMTB when the input checks are the bottleneck. This run-time improvement is due to improvements we added from sS13 and occurs in the keyed database, millionaires, and matrix multiplications programs. In the other program, edit distance, the input checks are not the bottleneck and PartialGC does not outperform CMTB. The total run-time increase for the edit distance problem is due to overhead of using the new sS13 OT cut-and-choose technique which requires sending each gate to the evaluator for check circuits and evaluation circuits. This is discussed further in Section 6.4. The typical use case we imagine for our system, however, is more like the keyed database program, which has a large amount of inputs and a very small circuit. We expand upon this use case later in this section.

	CMTB	PartialGC
KeyedDB 64	6,080	20,891
KeyedDB 128	12,160	26,971
KeyedDB 256	24,320	39,131
MatrixMult8x8	3,060,802	3,305,113
Edit Distance 128	1,434,888	1,464,490
Millionaires 8192	49,153	78,775
LCS Incremental 128	4,053,870	87,236
LCS Incremental 256	8,077,676	160,322
LCS Incremental 512	16,125,291	306,368
LCS Full 128	2,978,854	-
LCS Full 256	13,177,739	-

Table 1: Non-XOR gate counts for the various circuits. In the first 6 circuits, the difference between CMTB and PartialGC gate counts is in the consistency checks. The explanation for the difference in size between the incremental versions of longest common substring (LCS) is given in *Reusing Values*.

Reusing Values

For a test of our system's wire saving capabilities we tested a dynamic programming problem, longest common substring, in both PartialGC and CMTB. This program determines the length of the longest common substring between two strings. Rather than use a single computation for the solution, our version incrementally adds a single bit of input to both strings each time the computation is run and outputs the results each time to the evaluator. We believe this is a realistic comparison to a real-world application that incrementally adds data during each computation where it is faster to save the intermediate state and add to it after seeing an intermediate result than rerun the entire computation many times after seeing the result.

For our testing, PartialGC uses our technique to reuse wire values. In CMTB, we save each desired internal bit under a one-time pad and re-enter them into the next computation, as well as the information needed to decrypt the ciphertext. We use a MAC (the AES circuit of KSS12) to verify that the party saving the output bits did not modify them. We also use AES to generate a one-time pad inside the garbled circuit. We use AES as this is the only cryptographically secure function used in CMTB. Both parties enter private keys to the MAC functions. This program is labeled *CMTB-Inc*, for CMTB *incremental*. The size of this program represents the size of the total strings. We also created a circuit that computes the complete longest common substring in one computation labeled *CMTB-Full*.

The resulting size of the PartialGC and CMTB circuits are shown in Table 1, and the results are shown in Figure 4. This result shows that saving and reusing values in PartialGC is more efficient than completely rerunning the computation. The input consistency check adds considerably to the memory use on the phone for *CMTB-Inc* and in the case of input bit 512, the *CMTB-Inc* program will not complete. In the case of the 512-bit *CMTB-Full*, the program would not complete compilation in over 42 hours. In our *CMTB-Inc* program, we assume the cloud saves the output bits so that multiple phones can have a shared private key. We do

	16 Circuits			64 Circuits			256 Circuits		
	CMTB	PartialGC		CMTB	PartialGC		CMTB	PartialGC	
KeyedDB 64	18 ± 2%	3.5 ± 3%	5.1x	72 ± 2%	8.3 ± 5%	8.7x	290 ± 2%	26 ± 2%	11x
KeyedDB 128	33 ± 2%	4.4 ± 8%	7.5x	140 ± 2%	9.5 ± 4%	15x	580 ± 2%	31 ± 3%	19x
KeyedDB 256	65 ± 2%	4.6 ± 2%	14x	270 ± 1%	12 ± 6%	23x	1200 ± 3%	38 ± 5%	32x
MatrixMult8x8	48 ± 4%	46 ± 4%	1.0x	110 ± 8%	100 ± 7%	1.1x	400 ± 10%	370 ± 5%	1.1x
Edit Distance 128	21 ± 6%	22 ± 3%	0.95x	47 ± 7%	50 ± 9%	0.94x	120 ± 9%	180 ± 6%	0.67x
Millionaires 8192	35 ± 3%	7.3 ± 6%	4.8x	140 ± 2%	20 ± 2%	7.0x	580 ± 1%	70 ± 2%	8.3x

Table 2: Timing results comparing PartialGC to CMTB without saving any values. All times in seconds.

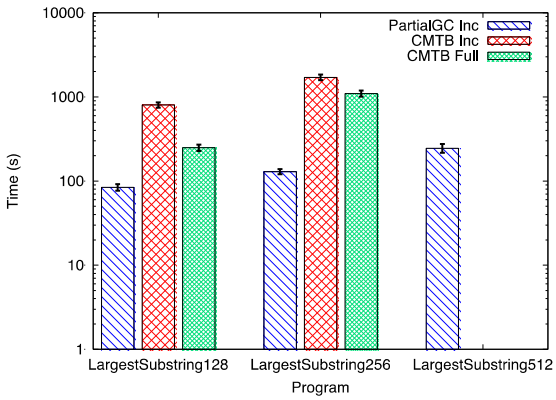


Figure 4: Results from testing our largest common substring (LCS) programs for PartialGC and CMTB. This shows when changing a single input value is more efficient under PartialGC than either CMTB program. CMTB crashed on running LCS Incremental of size 512 due to memory requirements. We were unable to complete the compilation of CMTB Full of size 512.

not provide a full program due to space requirements.

Note that the growth of *CMTB-Inc* and *CMTB-Full* are different. *CMTB-Full* grows at a larger rate (4x for each 2x factor increase) than *CMTB-Inc* (2x for each 2x factor increase), implying that although at first it seems more efficient to rerun the program if small changes are desired in the input, eventually this will not be the case. Even with a more efficient AES function, *CMTB-Inc* would not be faster as the bottleneck is the input, not the size of the circuit.

The overhead of saving and reusing values is discussed further in Appendix B.

Outsourcing to a Server Process

PartialGC can be used in other scenarios than just outsourcing to a mobile device. It can outsource garbled circuit evaluation from a single server process and retain performance benefits over a single server process of CMTB. For this experiment the outsourcing party has a single thread. Table 4 displays these results and shows that in the KeyedDB 256 program, PartialGC has a 92% speedup over CMTB. As with the outsourced mobile case, keyed database problems perform particularly well in PartialGC. Because the computationally-intensive input consistency check is a greater bottleneck on mobile devices than servers, these improvements for most programs are less dramatic. In particular, both edit distance and matrix multiplication programs benefit from higher computational power and their bottlenecks on a server are no longer input consistency; as a result, they execute faster in CMTB than in PartialGC.

	256 Circuits		
	CMTB	PartialGC	
KeyedDB 64	64992308	3590416	18x
KeyedDB 128	129744948	3590416	36x
KeyedDB 256	259250228	3590416	72x
MatrixMult8x8	71238860	35027980	2.0x
Edit Distance 128	2615651	4108045	0.64x
Millionaires 8192	155377267	67071757	2.3x

Table 3: Bandwidth comparison of CMTB and PartialGC. Bandwidth counted by instrumenting PartialGC to count the bytes it was sending and receiving and then adding them together. Results in bytes.

6.2 Bandwidth

Since the main reason for outsourcing a computation is to save on resources, we give results showing a decrease in the evaluator's bandwidth. Bandwidth is counted by making the evaluator to count the number of bytes PartialGC sends and receives to either server. Our best result gives a 98% reduction in bandwidth (see Table 3). For the edit distance, the extra bandwidth used in the outsourced oblivious transfer for all circuits, instead of only the evaluation circuits, exceeds any benefit we would otherwise have received.

6.3 Secure Friend Finder

Many privacy-preserving applications can benefit from using PartialGC to cache values for state. As a case study, we developed a privacy-preserving friend finder application, where users can locate nearby friends without any user divulging their exact location. In this application, many different mobile phone clients use a consistent generator (a server application) and outsource computation to a cloud. The generator must be the same for all computations; the cloud must be the same for each computation. The cloud and generator are two different parties. After each computation, the map is updated when PartialGC saves the current state of the map as wire labels. Without PartialGC outsourcing values to the cloud, the wire labels would have to be transferred directly between mobile devices, making a multi-user application difficult or impossible.

We define three privacy-preserving operations that comprise the application's functionality:

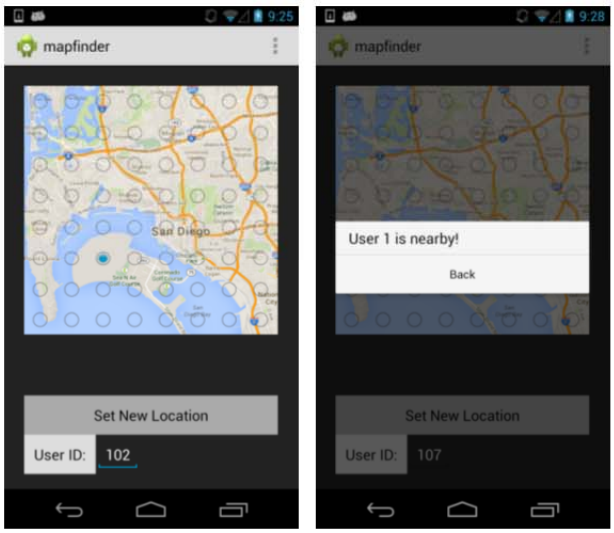
MapStart - The three parties (generator, evaluator, cloud) create a "blank" map region, where all locations in the map are blank and remain that way until some mobile party sets a location to his or her ID.

MapSet - The mobile party sets a single map cell to a new value. This program takes in partial values from the generator and cloud and outputs a location selected by the mobile party.

MapGet - The mobile party retrieves the contents of a single map cell. This program retrieves partial values from the generator and cloud and outputs any ID set for that cell to the mobile.

	16 Circuits			64 Circuits			256 Circuits		
	CMTB	PartialGC		CMTB	PartialGC		CMTB	PartialGC	
KeyedDB 64	6.6 ± 4%	1.4 ± 1%	4.7x	27 ± 4%	5.1 ± 2%	5.3x	110 ± 2%	24.9 ± 0.3%	4.4x
KeyedDB 128	13 ± 3%	1.8 ± 2%	7.2x	54 ± 4%	5.8 ± 2%	9.3x	220 ± 5%	27.9 ± 0.5%	7.9x
KeyedDB 256	25 ± 4%	2.5 ± 1%	10x	110 ± 7%	7.3 ± 2%	15x	420 ± 4%	33.5 ± 0.6%	13x
MatrixMult8x8	42 ± 3%	41 ± 4%	1.0x	94 ± 4%	79 ± 3%	1.2x	300 ± 10%	310 ± 1%	0.97x
Edit Distance 128	18 ± 3%	18 ± 3%	1.0x	40 ± 8%	40 ± 6%	1.0x	120 ± 9%	150 ± 3%	0.8x
Millionaires 8192	13 ± 4%	3.2 ± 1%	4.1x	52 ± 3%	8.5 ± 2%	6.1x	220 ± 5%	38.4 ± 0.9%	5.7x

Table 4: Timing results from outsourcing the garbled circuit evaluation from a single server process. Results in seconds.



(a) Location selected. (b) After computation.

Figure 5: Screenshots from our application. (a) shows the map with radio buttons a user can select to indicate position. (b) show the result after "set new position" is pressed when a user is present. The application is set to use 64 different map locations. Map image from Google Maps.

In the application, each user using the *Secure Friend Finder* has a unique ID that represents them on the map. We divide the map into 'cells', where each cell is a set amount of area. When the user presses "Set New Location', the program will first look to determine if that cell is occupied. If the cell is occupied, the user is informed he is near a friend. Otherwise the cell is updated to contain his user ID and remove his ID from his previous location. We assume a maximum of 255 friends in our application since each cell in the map is 8 bits.

Figure 6 shows the performance of these programs in the malicious model with a 2^{-80} security parameter (evaluated over 256 circuits). We consider map regions containing both 256 and 2048 cells. For maps of 256 cells, each operation takes about 30 seconds.[1] As there are three operations for each "Set New Location" event, the total execution time is about 90 seconds, while execution time for 2048 cells is about 3 minutes. The bottleneck of the 64 and 256 cell maps is the outsourced oblivious transfer, which is not affected by the number of cells in the map. The vastly larger circuit associated with the 2048-cell map makes getting and setting values slower operations, but these results show such an application is practical for many scenarios.

Example - As an example, two friends initiate a friend finder computation using Amazon as the cloud and Facebook as the generator. The first friend goes out for a coffee at a café. The second friend, riding his bike, gets a message that his friend is nearby and looks for a few minutes and finds him in the café. Using this application prevents either Amazon or Facebook from knowing either user's location while they are able to learn whether they are nearby.

[1]Our 64-cell map, as seen the application screenshots, also takes about 30 seconds for each operation.

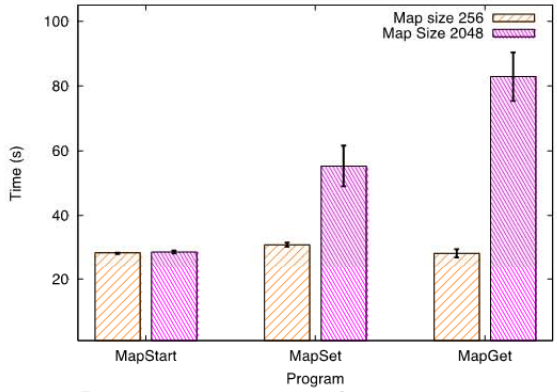


Figure 6: Run time comparison of our map programs with two different map sizes.

6.4 Discussion

Analysis of improvements

We analyzed our results and found the improvements came from three places: the improved sS13 consistency check, the saving and reusing of values, and the fixed oblivious transfer. In the case of the sS13 consistency check, there are two reasons for the improvement, first there is less network traffic and second it does not use exponentiations. In the case of saving and reusing values, we save time by the faster input consistency check and not requiring a user to recompute a circuit multiple times. Lastly, we reduced the runtime and bandwidth by fixing parts of the OOT. The previous outsourced oblivious transfer performed the primitive OT S times instead of a single time, which turn forced many extra exponentiations. Each amount of improvement varies depending upon the circuit.

Output check

Although the garbled circuit is larger for our output check, this check performs less cryptographic operations for the outsourcing party, as the evaluator only has to perform a MAC on the output of the garbled circuit. We use this check to demonstrate using a MAC can be an efficient output check for a low power device when the computational power is not equivalent across all parties.

Commit Cut-and-Choose vs OT Cut-and-Choose

Our results unexpectedly showed that the sS13 OT cut-and-choose used in PartialGC is actually slower than the KSS12 commit cut-and-choose used in CMTB in our experimental setup. Theoretically, sS13, which requires fewer

cryptographic operations, as it generates the garbled circuit only once, should be the faster protocol. The difference between the two cut-and-choose protocols is the network usage – instead of $\frac{2}{5}$ of the circuits (CMTB), *all* the circuits must be transmitted in sS13. The sS13 cut-and-choose is required in our protocol so that the cloud can check that the generator creates the correct gates.

7. RELATED WORK

SFE was first described by Yao in his seminal paper [39] on the subject. The first general purpose platform for SFE, Fairplay [32], was created in 2004. Fairplay had both a compiler for creating garbled circuits, and a run-time system for executing them. Computations involving three or more parties have also been examined; one of the earliest examples is FairplayMP [2]. There have been multiple other implementations since, in both semi-honest [6, 9, 16, 17, 40] and malicious settings [26, 37].

Optimizations for garbled circuits include the free-XOR technique [25], garbled row reduction [36], rewriting computations to minimize SFE [23], and pipelining [18]. Pipelining allows the evaluator to proceed with the computation while the generator is creating gates.

KSS12 [27] included both an optimizing compiler and an efficient run-time system using a parallelized implementation of SFE in the malicious model from [37].

The creation of circuits for SFE in a fast and efficient manner is one of the central problems in the area. Previous compilers, from Fairplay to KSS12, were based on the concept of creating a complete circuit and then optimizing it. PAL [33] improved such systems by using a simple template circuit, reducing memory usage by orders of magnitude. PCF [26] built from this and used a more advanced representation to reduce the disk space used.

Other methods for performing MPC involve homomorphic encryption [3, 12], secret sharing [4], and ordered binary decision diagrams [28]. A general privacy-preserving computation protocol that uses homomorphic encryption and was designed specifically for mobile devices can be found in [7]. There are also custom protocols designed for particular privacy-preserving computations; for example, Kamara et al. [21] showed how to scale server-aided Private Set Intersection to billion-element sets with a custom protocol.

Previous reusable garbled-circuit schemes include that of Brandão [5], which uses homomorphic encryption, Gentry *et al.* [10], which uses attribute-based functional encryption, and Goldwasser *et al.* [13], which introduces a succinct functional encryption scheme. These previous works are purely theoretical; none of them provides experimental performance analysis. There is also recent theoretical work on reusing encrypted garbled-circuit values [30, 11, 31] in the ORAM model; it uses a variety of techniques, including garbled circuits and identity-based encryption, to execute the underlying low-level operations (program state, read/write queries, etc.). Our scheme for reusing encrypted values is based on completely different techniques; it enables us to do new kinds of computations, thus expanding the set of things that can be computed using garbled circuits.

The Quid-Pro-Quo-tocols system [19] allows fast execution with a single bit of leakage. The garbled circuit is executed twice, with the parties switching roles in the latter execution, then running a secure protocol to ensure that the output from both executions are equivalent; if this fails, a single bit may be leaked due to the selective failure attack.

8. CONCLUSION

This paper presents PartialGC, a server-aided SFE scheme allowing the reuse of encrypted values to save the costs of input validation and to allow for the saving of state, such that the costs of multiple computations may be amortized. Compared to the server-aided outsourcing scheme by CMTB, we reduce costs of computation by up to 96% and bandwidth costs by up to 98%. Future work will consider the generality of the encryption re-use scheme to other SFE evaluation systems and large-scale systems problems that benefit from the addition of state, which can open up new and intriguing ways of bringing SFE into the practical realm.

Acknowledgements: This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory under contracts FA8750-11-2-0211 and FA8750-13-2-0058. It is also supported in part by the U.S. National Science Foundation under grant numbers CNS-1118046 and CNS-1254198. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, NSF, or the U.S. Government.

9. REFERENCES

[1] M. Bellare and S. Micali. Non-Interactive Oblivious Transfer and Applications. In *Proceedings of CRYPTO*, 1990.

[2] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *Proceedings of the ACM conference on Computer and Communications Security*, 2008.

[3] R. Bendlin, I. Damgård, C. Orlandi, and S. Zakarias. Semi-Homomorphic Encryption and Multiparty Computation. In *Proceedings of EUROCRYPT*, 2011.

[4] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security - ESORICS'08*, 2008.

[5] L. T. A. N. Brandão. Secure Two-Party Computation with Reusable Bit-Commitments, via a Cut-and-Choose with Forge-and-Lose Technique. Technical report, University of Lisbon, 2013.

[6] M. Burkhart, M. Strasser, D. Many, and X. Dimitropoulos. Sepia: Privacy-preserving aggregation of multi-domain network events and statistics. In *Proceedings of the 19th USENIX Conference on Security*, USENIX Security'10, pages 15–15, Berkeley, CA, USA, 2010. USENIX Association.

[7] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. In *Journal of Security and Communication Networks (SCN)*, To appear 2014.

[8] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for

mobile devices. In *Proceedings of the USENIX Security Symposium*, 2013.

[9] I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Proceedings of the 12th International Conference on Practice and Theory in Public Key Cryptography: PKC '09*, Irvine, pages 160–179, Berlin, Heidelberg, 2009. Springer-Verlag.

[10] C. Gentry, S. Gorbunov, S. Halevi, V. Vaikuntanathan, and D. Vinayagamurthy. How to compress (reusable) garbled circuits. Cryptology ePrint Archive, Report 2013/687, 2013. `http://eprint.iacr.org/`.

[11] C. Gentry, S. Halevi, S. Lu, R. Ostrovsky, M. Raykova, and D. Wichs. Garbled ram revisited. In *Advances in Cryptology–EUROCRYPT 2014*, pages 405–422. Springer Berlin Heidelberg, 2014.

[12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic Evaluation of the AES Circuit. In *Proceedings of CRYPTO*, 2012.

[13] S. Goldwasser, Y. Kalai, R. A. Popa, V. Vaikuntanathan, and N. Zeldovich. Reusable Garbled Circuits and Succinct Functional Encryption. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, STOC '13, 2013.

[14] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Proceedings of the theory and applications of cryptographic techniques annual international conference on Advances in cryptology*, 2008.

[15] S. Halevi, Y. Lindell, and B. Pinkas. Secure Computation on the Web: Computing without Simultaneous Interaction. In *CRYPTO'11*, 2011.

[16] W. Henecka, S. Kögl, A.-R. Sadeghi, T. Schneider, and I. Wehrenberg. Tasty: tool for automating secure two-party computations. In *Proceedings of the ACM conference on Computer and Communications Security*, 2010.

[17] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 772–783, New York, NY, USA, 2012. ACM.

[18] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[19] Y. Huang, J. Katz, and D. Evans. Quid-Pro-Quo-tocols: Strengthening Semi-Honest Protocols with Dual Execution. *IEEE Symposium on Security and Privacy*, (33rd), May 2012.

[20] Y. Ishai, J. Kilian, K. Nissim, and E. Petrank. Extending oblivious transfers efficiently. In *Proceedings of CRYPTO*, 2003.

[21] S. Kamara, P. Mohassel, M. Raykova, and S. Sadeghian. Scaling private set intersection to billion-element sets. Technical Report MSR-TR-2013-63, Microsoft Research, 2013.

[22] S. Kamara, P. Mohassel, and B. Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[23] F. Kerschbaum. Expression rewriting for optimizing secure computation. In *Conference on Data and Application Security and Privacy*, 2013.

[24] M. S. Kiraz and B. Schoenmakers. A protocol issue for the malicious case of yao's garbled circuit construction. In *Proceedings of Symposium on Information Theory in the Benelux*, 2006.

[25] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the international colloquium on Automata, Languages and Programming, Part II*, 2008.

[26] B. Kreuter, B. Mood, a. shelat, and K. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *Proceedings of the USENIX Security Symposium*, 2013.

[27] B. Kreuter, a. shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the USENIX Security Symposium*, 2012.

[28] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure function evaluation with ordered binary decision diagrams. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2006.

[29] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the annual international conference on Advances in Cryptology*, 2007.

[30] S. Lu and R. Ostrovsky. How to garble ram programs. In *Advances in Cryptology–EUROCRYPT 2013*, pages 719–734. Springer Berlin Heidelberg, 2013.

[31] S. Lu and R. Ostrovsky. Garbled ram revisited, part ii. Cryptology ePrint Archive, Report 2014/083, 2014. `http://eprint.iacr.org/`.

[32] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–a secure two-party computation system. In *Proceedings of the USENIX Security Symposium*, 2004.

[33] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[34] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *Proceedings of the annual ACM Symposium on Theory of Computing (STOC)*, 1999.

[35] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the annual ACM-SIAM Symposium on Discrete algorithms (SODA)*, 2001.

[36] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation is Practical. In *ASIACRYPT*, 2009.

[37] a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *Proceedings of EUROCRYPT*, 2011.

[38] a. shelat and C.-H. Shen. Fast two-party secure computation with minimal assumptions. In *Conference on Computer and Communications Security (CCS)*, 2013.

[39] A. C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[40] Y. Zhang, A. Steele, and M. Blanton. PICCO: A General-purpose Compiler for Private Distributed Computation. In *Proceedings of the ACM Conference on Computer Communications Security (CCS)*, 2013.

APPENDIX

A. CMTB PROTOCOL

As we are building off of the CMTB garbled circuit execution system, we give an abbreviated version of the protocol. In our description we refer to the generator, the cloud, and the evaluator. The cloud is the party the evaluator outsources her computation to.

Circuit generation and check: The template for the garbled circuit is augmented to add one-time XOR pads on the output bits and split the evaluator's input wires per the input encoding scheme. The generator generates the necessary garbled circuits and commits to them and sends the commitments to the evaluator. The generator then commits to input labels for the evaluator's inputs.

CMTB relies on Goyal et al.'s [14] random seed check, which was implemented by Kreuter et al. [27] to combat generation of incorrect circuits. This technique uses a cut-and-choose style protocol to determine whether the generator created the correct circuits by creating and committing to many different circuits. Some of those circuits are used for evaluation, while the others are used as check circuits.

Evaluator's inputs: Rather than a two-party oblivious transfer, we perform a three-party *outsourced oblivious transfer*. An outsourced oblivious transfer is an OT that gets the select bits from one party, the wire labels from another, and returns the selected wire labels to a third party. The party that selects the wire labels does not learn what the wire labels are, and the party that inputs the wire labels does not learn which wire was selected; the third party only learns the selected wire labels. In CMTB, the generator offers up wire labels, the evaluator provides the select bits, and the cloud receives the selected labels. CMTB uses the Ishai OT extension [20] to reduce the number of OTs.

CMTB uses an encoding technique from Lindell and Pinkas [29], which prevents the generator from finding out any information about the evaluator's input if a selective failure attack transpires. CMTB also uses the commitment technique of Kreuter et al. [27] to prevent the generator from swapping the two possible outputs of the oblivious transfer. To ensure the evaluator's input is consistent across all circuits, CMTB uses a technique from Lindell and Pinkas [29], whereby the inputs are derived from a single oblivious transfer.

Generator's input and consistency check: The generator sends his input to the cloud for the evaluation circuits. Then the generator, evaluator, and cloud all work together to prove the input consistency of the generator's input. For the generator's input consistency check, CMTB uses the malleable-claw free construction from shelat and Shen [37].

Circuit evaluations: The cloud evaluates the garbled circuits marked for evaluation and checks the circuits marked for checking. The cloud enters in the generator and evaluator's input into each garbled circuit and evaluates each circuit. The output for any particular bit is then the majority output between all evaluator circuits. The cloud then recreates each check circuit. The cloud creates the hashes of each garbled circuit and sends those hashes to the evaluator. The evaluator then verifies the hashes are the same as the ones the generator previously committed to.

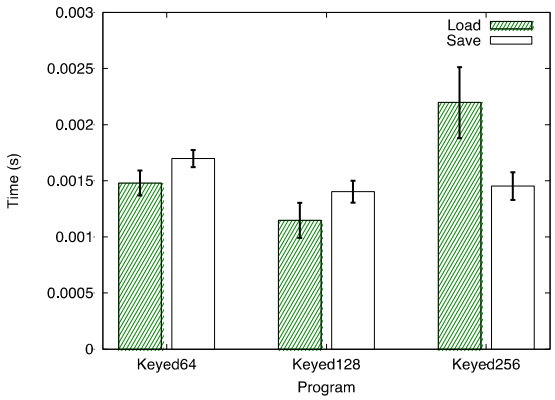


Figure 7: The amount of time it takes to save and load a bit in PartialGC when using 256 circuits.

Output consistency check and output: The three parties prove together that the cloud did not modify the output before she sent it to the generator or evaluator. Both the evaluator and generator receive their respective outputs. All outputs are blinded by the respective party's one-time pad inside the garbled circuit to prevent the cloud from learning what any output bit represents.

CMTB uses the XOR one-time pad technique from Kiraz [24] to prevent the evaluator from learning the generator's real output. To prevent output modification, CMTB uses the witness-indistinguishable zero-knowledge proof from Kreuter et al. [27].

B. OVERHEAD OF REUSING VALUES

We created several versions of the keyed database program to determine the runtime of saving and loading the database on a per bit basis using our system (See Figure 7). This figure shows it is possible to save and load a large amount of saved wire labels in a relatively short time. The time to load a wire label is larger than the time to save a value since saving only involves saving the wire label to a file and loading involves reading from a file and creating the partial input gates. Although not shown in the figure, the time to save or load a single bit also increases with the circuit parameter. This is because we need S copies of that bit - one for every circuit.

C. EXAMPLE PROGRAM

In this section we describe the execution of an *attendance application*. Imagine a building where the host wants each user to sign in from their phones to keep a log of the guests, but also wants to keep this information secret.

This application has three distinct programs. The first program initializes a counter to a number input by the evaluator. The second program, which is used until the last program is called, takes in a name and increments the counter by one. The last program outputs all names and returns the count of users.

For this application, users (rather, their mobile phones) assume the role of evaluators in the protocol (Section 4).

First, the host runs the initial program to initialize a

database. We cannot execute the second program to add names to the log until this is done, lest we reveal that there is no memory saved (*i.e.*, there is no one else present).

Protocol in Brief: In this first program, the cut-and-choose OT is executed to select the circuit split (the circuits that are for evaluation and generation). Both parties save the decryption keys: the cloud saves the keys attained from the OT and the generator saves both possible keys that could have been selected by the cloud. The evaluator performs the OOT with the other parties to input the initial value into the program. There is no input by the generator so the generator's input check does not execute. There is no partial input so that phase of the protocol is skipped. The garbled circuit to set the initial value is executed; while there is no output to the generator or evaluator, a partial output is produced: the cloud saves the garbled wire value, which it possesses, and the generator saves both possible wire values (the generator does not know what value the cloud has, and the cloud does not know what the value it has saved actually represents). The cloud also saves the circuit split.

Saved memory after the program execution (when the evaluator inputs 0 as the initial value):

Count
0
Saved Guests

Guest 1 then enters the building and executes the program, entering his name ("Guest 1") as input.

Protocol in Brief: In this second program, the cut-and-choose OT is not executed. Instead, both the generator and cloud load the saved decryption key values, hash them, and use those values for the check and evaluation circuit information (instead of attaining new keys through an OT, which would break security). The new keys are saved, and the evaluator then performs the OOT for input. The generator does not have any input in this program so the check for the generator's input is skipped. Since there exists a partial input, the generator loads both possible wire values and creates the partial input gates. The cloud loads the attained values, receives the partial input gates from the generator, and then executes (and checks) the partial input gates to receive the garbled input values. The garbled circuit is then executed and partial output saved as before (although there is more data to save for this program as there is a name present in the database).

After executing the second program the memory is as follows:

Count
1
Saved Guests
Guest1

Guest 2 then enters the dwelling and runs the program. The execution is similar to the previous one (when Guest 1 entered), except that it's executed by Guest 2's phone.

At this point, the memory is as follows:

Count
2
Saved Guests
Guest1
Guest2

Guest 3 then enters the dwelling and executes the program as before. At this point, the memory is as follows:

Count
3
Saved Guests
Guest1
Guest2
Guest3

Finally, the host runs the last program that outputs the count and the guests in the database. In this case the count is 3 and the guests are *Guest*1, *Guest*2, and *Guest*3.

Whitewash: Outsourcing Garbled Circuit Generation for Mobile Devices

Henry Carter
Georgia Institute of Technology
carterh@gatech.edu

Charles Lever
Georgia Institute of Technology
chazlever@gatech.edu

Patrick Traynor
University of Florida
traynor@cise.ufl.edu

ABSTRACT

Garbled circuits offer a powerful primitive for computation on a user's personal data while keeping that data private. Despite recent improvements, constructing and evaluating circuits of any useful size remains expensive on the limited hardware resources of a smartphone, the primary computational device available to most users around the world. In this work, we develop a new technique for securely outsourcing the generation of garbled circuits to a Cloud provider. By outsourcing the circuit generation, we are able to eliminate the most costly operations from the mobile device, including oblivious transfers. Our proofs of security show that this technique provides the best security guarantees of any existing garbled circuit outsourcing protocol. We also experimentally demonstrate that our new protocol, on average, decreases execution time by 75% and reduces network costs by 60% compared to previous outsourcing protocols. In so doing, we demonstrate that the use of garbled circuits on mobile devices can be made nearly as practical as it is becoming for server-class machines.

1. INTRODUCTION

Mobile devices have become one of the dominant computing platforms, with approximately 57% market penetration in the United States alone [9]. These devices are capable of gathering and storing all of a user's personal data, from current location and social contacts to banking and electronic payment information. Because of the personal nature of these devices, it is critical that a user's information be protected at all times. Unfortunately, many smartphone applications that require users to send data to application servers make preserving the privacy of this data difficult.

To resolve this issue, a variety of secure multiparty computation techniques exist that could be leveraged to perform computation over encrypted inputs [6, 11, 12, 28]. Currently, the most practically efficient two-party technique is the Yao Garbled Circuit [43]. Despite recent improvements in the efficiency of garbled circuits [27, 41], this technique still requires significant computation and communication resources, rendering it impractical for most smartphones. One possible solution to this imbalance of resources is to blindly outsource the heavy computation to the Cloud. However, because of the untrusted nature of Cloud providers [42], such a solution fails to provide measurable guarantees for applications requiring high assurance.

In this work, we develop a new protocol for securely outsourcing garbled circuit generation to an untrusted Cloud. We construct a protocol that offloads the role of generating the garbled circuit from the mobile device to the Cloud without exposing any private inputs or outputs. By choosing to outsource this portion of the protocol, we eliminate a significant number of expensive public-key cryptography operations and rounds of communication used in oblivious transfer. The result is a more computationally and bandwidth efficient outsourcing protocol with the strongest security guarantees of any outsourcing technique to date [8, 23].

In this paper, we make the following contributions:

- **Develop a new outsourcing protocol:** We develop the Whitewash[1] outsourcing protocol, which allows a mobile device participating in a two-party secure function evaluation to outsource the generation of the garbled circuit. Our protocol assigns the mobile device the role of circuit generator instead of circuit evaluator, outsourcing a completely different set of operations from previous outsourcing protocols [8, 23]. By reversing the functions of the two players, we fully eliminate the requirement for any oblivious transfers, outsourced or otherwise, to or from the mobile device. This "simple" role reversal requires fundamentally redesigning the outsourcing techniques used in previous work, as well as new security proof formulations.
- **Formal verification and analysis:** We formally prove the security of our outsourcing techniques in the malicious model defined by Kamara et al. [23]. Unlike previous work [8, 23], our protocol provides security when the mobile device is colluding with its Cloud provider against the application server. *This added security guarantee makes our protocol the most secure outsourcing protocol to date.* We then provide an analysis of the reduction in operations between our work and the outsourced oblivious transfer of Carter et al. [8], as well as the Salus framework by Kamara et al. [23]. Specifically, our protocol requires more executions of a pseudorandom number generator in exchange for fewer algebraic group operations and zero-knowledge proofs. Moreover, we significantly reduce the number of rounds of communication required to the mobile device.
- **Implement and evaluate the performance of our protocol:** In our performance evaluation, we demonstrate a maximum improvement of 98% in execution time and 92% improvement in bandwidth overhead compared to Carter et al. [8] (with 75% and 60% average improvement, respectively). For a different

[1]A reference to Tom Sawyer, who "outsourced" his chores to his friends without ever revealing the true nature of the work.

ACSAC '14 December 08 - 12 2014, New Orleans, LA, USA

ACM 978-1-4503-3005-3/14/12..$15.00
http://dx.doi.org/10.1145/2664243.2664255

test application, when compared to performing computation directly on the mobile device [28], we demonstrated a 96% and 90% improvement in execution time and bandwidth, respectively. These improvements allow for the largest circuits evaluated on any platform to be computed from a mobile device efficiently and with equivalent security parameters to non-mobile protocols.

The rest of this work is organized as follows: Section 2 provides detail on related research; Section 3 describes our threat model and security definition; Section 4 provides a description of the Whitewash protocol; Section 5 compares the operations required in our protocol to the protocols by Carter et al. and Kamara et al.; Section 6 describes our empirical performance analysis; and Section 7 provides concluding remarks.

2. RELATED WORK

Fairplay [35] provided the first practically efficient implementation of Yao's garbled circuit protocol [43], requiring only simple hash and symmetric key operations to securely evaluate an arbitrary function. Since then, a variety of garbled circuit-based secure function evaluation (SFE) protocols have been developed in the semi-honest adversarial model [5, 16, 20, 21, 29, 31, 34, 39]. The latest of these, developed by Huang et al. [16], allows garbled circuits to be evaluated in stages, which makes it the most efficient semi-honest garbled circuit evaluation technique, both in computation and memory requirements. In recent work, several garbled circuit SFE protocols have been developed in the malicious security model, which require significantly more computational resources than semi-honest protocols, but are secure against arbitrary polynomial-time adversaries [18, 26, 28, 30, 33, 37, 40]. The protocol by shelat and Shen [41] provides a two-party garbled circuit protocol which uses only symmetric-key constructions outside of the oblivious transfer. When combined with Huang's pipelining approach and the PCF compiler by Kreuter et al. [27], their protocol is among the most efficient maliciously-secure garbled circuit protocols implemented to date. Some efforts have been made to improve the efficiency of these protocols by slightly reducing the adversary model. Many schemes have been developed in the covert adversary model, which allows for some efficiency gains at the cost of security [2, 10, 15, 36]. Huang et al. [17] developed a protocol that leaks only one bit of input to a malicious adversary through dual execution, which was later implemented on GPUs by Husted et al. [19]. In order to further improve the efficiency of garbled circuit protocols, Gordon et al. [13] developed a protocol that combined Oblivious RAM with garbled circuits, allowing sub-linear amortized complexity. However, this protocol only allows this performance gain for functions that can be computed efficiently on a random-access machine. Other work has focused on making circuit compilation and garbling more efficient [4, 38]. These techniques improve the underlying operations found in all garbled circuit execution protocols, providing improvement in all existing techniques.

To further improve the speed of cryptographic protocols on devices with minimal computational resources, the idea of outsourcing cryptographic operations has been explored for many years in the field of Server-assisted cryptography [3]. More recently, Green et al. [14] developed a technique for outsourcing the costly decryption of attribute-based encryption schemes to the cloud without revealing the contents of the ciphertext. Atallah and Frikken [1] developed a set of special-purpose protocols for securely outsourcing Linear Algebra computations to a single cloud server. For data mining applications, Kerschbaum recently developed an outsourced set intersection protocol using homomorphic encryption techniques [24]. While all of these applications provide significant performance gains for specific cryptographic applications, none of them address outsourcing of general secure computation.

In their Salus protocol, Kamara et al. [22, 23] developed two protocols for securely outsourcing the computation of arbitrary functions to the cloud. Following Salus, Carter et al. [8] developed an outsourcing protocol based on the maliciously secure garbled circuit protocol by Kreuter et al. [28]. Carter's protocol outsources the evaluation of garbled circuits by adding in an Outsourced Oblivious Transfer primitive. Their participant configuration is the same configuration found in Kamara's maliciously secure protocol, where the cloud is made responsible for evaluating the garbled circuit. In this work, we choose to build on shelat and Shen's latest protocol [41] since the symmetric execution environment of Huang et al. [18] does not lend itself to outsourcing, and the bootstrapping technique used by Lindell [30] has not been implemented or evaluated in practice. Unlike previous work, we choose to fundamentally rearrange the roles of the participants, outsourcing the generation of the garbled circuits as in Kamara's covertly secure protocol.

3. OVERVIEW AND DEFINITIONS

3.1 Protocol Goals and Summary

The primary reason for developing an outsourcing protocol for secure function evaluation is to allow two parties of asymmetrical computing ability to securely compute some result. Current two-party computation protocols assume both parties are equipped with equivalent computing resources and so require both parties to perform comparable operations. However, when a mobile device is taking part in computation with an application server, some technique is necessary to reduce the complexity of the operation on the mobile device. Ideally, we can make the mobile device perform some small number of operations that is independent of the size of the circuit being evaluated.

In constructing such a protocol, there are four goals that we would like to satisfy. The first of these goals is correctness. It is necessary that an outsourcing protocol must produce correct output even in the face of malicious players attempting to corrupt the computation. The second desirable guarantee is security. SFE protocols frequently use a simulation-based approach to defining and proving security, which we outline in detail below. Essentially, the goal is to show that each party can learn the output of the computed function and nothing else. Third, an ideal protocol would provide some guarantee of fair release. This guarantee ensures that either both parties receive their outputs from the computation, or neither party receives their output. Our protocol achieves this in all but one corruption scenario by treating the Cloud as an arbiter, who will simultaneously and fairly release the outputs of the protocol using one-time pads. In the scenario where the mobile device and Cloud are colluding, it is possible for the Cloud to terminate the protocol after the mobile device receives output but before the application server receives output. However, this is inherently possible in most two-party garbled circuit protocols. The fourth goal of our protocol is efficiency. Our outsourcing protocol balances a minimal set of operations on the mobile device with efficient multiparty computation operations on the application server and Cloud.

Given these goals, we build our protocol in the two-party server-aided multiparty computation setting. This setting is composed of two parties, the mobile device and an application server, who wish to run a two-party secure computation while keeping both of their inputs private. To assist the mobile device, the server-aided setting adds a third party Cloud provider, which is independent and non-colluding with the application server (more on non-collusion in the following section). The Cloud performs cryptographic operations

for the mobile device, but is not allowed to see any party's input or output from the computation.

To achieve our goals in this setting, we first select the most efficient two-party garbled circuit computation protocol to date that provides guarantees of correctness and security in the malicious model. We assign the mobile device the role of circuit generator in this protocol, and the application server is assigned the role of circuit evaluator. To outsource the circuit generation operations from the mobile device, we allow the device to generate short random seeds and pass these values to a Cloud computation provider, which then generates the garbled circuits using these seeds to generate randomness. Thus, the mobile device's work is essentially reduced to (1) generating random strings on the order of a statistical security parameter, and (2) garbling and sending its input values to the evaluating party. In this way, we develop a secure computation protocol where the mobile device performs work that is independent of the size of the function being evaluated. We provide a formal proof of security for our protocol in our technical report [7].

3.2 Non-collusion

To maintain security, previous outsourcing protocols assume that neither party colludes with the Cloud [8, 23]. The theoretical intuition for this constraint, outlined by Kamara et al. [23], is that the existence of an outsourcing protocol where parties can arbitrarily collude would imply a two-party secure multiparty computation protocol where one party performs sub-linear work with respect to the size of the circuit. While this has been shown to be possible using fully homomorphic encryption and, in some cases, oblivious RAM [13], it is not clear that these techniques can be efficiently applied to a garbled circuit outsourcing scheme. Because of this, previous work has left the more complex security model for future study. However, while previous protocols restrict collusion between the Cloud and any party, the sub-linear work implication only applies to cases when the Cloud is generating circuits and colludes with the evaluating party, or vice versa. In the Whitewash protocol, we prove security when the mobile device colludes with the Cloud against the evaluating web application. While this collusion scenario removes the fair release guarantee of our protocol, it in no way compromises the security guarantees of confidentiality of participant's inputs and outputs. Essentially, it reduces to the two-party computation scenario that the underlying protocol is proven to be secure in. Since the mobile device is paying the Cloud for computation services, we believe it is a more realistic assumption to assume that a Cloud provider could collude maliciously with the paying customer, and note that our protocol is the first outsourcing protocol to provide any security guarantees in the face of collusion with the Cloud.

3.3 Security Constructions

In the two-party computation protocol underlying our work, shelat and Shen implement a number of new and efficient cryptographic checks to ensure that none of the parties participating in the computation behave maliciously. We provide an overview of these security checks in the following section. We refer the reader to shelat and Shen's work for more formal definitions and proofs [41].

3.3.1 k-probe-resistant input encoding

When working with garbled circuit protocols in the malicious model, the generator has the ability to learn information about the evaluator's input by corrupting the wire labels sent during the oblivious transfer. This attack, known as selective failure, was first proposed by Mohassel and Franklin [37] as well as Kiraz and Schoenmakers [25]. In the server-aided setting, it is possible that the mobile device and the Cloud could collude and carry out this attack to recover the application server's input. To prevent this attack, shelat and Shen [41] implement an improved version of the k-probe-resistant input encoding mechanism originally proposed by Lindell and Pinkas [32]. In their protocol, the evaluator does not input her real input y to the computation, but chooses her input $\overline{y}$ such that $\mathbf{M} \cdot \overline{y} = y$ for a k-probe resistant matrix $\mathbf{M}$. Intuitively, the idea is that the generator would have to probe the evaluator's input approximately 2^k times before learning anything about her input y.

3.3.2 2-Universal Hash Function

A second concern with garbled circuits in the malicious model is that the generator may send different input values for each of the evaluated circuits from the cut-&-choose. As in the two-party setting, it is possible for the mobile device to submit inconsistent inputs to the application server in the server-aided setting. To ensure that the generator's inputs are consistent across evaluation circuits, shelat and Shen implement an efficient witness-indistinguishable proof, which computes a randomized, 2-universal hash of the input value using only arithmetic operations on matrices. Because of the regularity guarantees of a 2-universal hash, the outputs of these hash operations can be seen by the evaluator without revealing any information about the generator's inputs. However, if any of the hashed input values is inconsistent across evaluation circuits, the evaluator can infer that the generator provided inconsistent inputs, and can terminate the protocol.

3.3.3 Output proof of consistency

When a function being evaluated using garbled circuits has separate, private outputs for the generating and evaluating parties, it is necessary to ensure that the evaluating party does not tamper with the generating party's output. Since the output must be decoded from the garbled output wires for the majority check at the end of the protocol, if the output is only blinded with a one-time pad, this allows the evaluator the opportunity to change bits of the generator's output. Our setting faces the same potential for attack from the application server, who is responsible for evaluating the circuit and distributing the blinded output. Several techniques for preventing this kind of tampering have been proposed, but shelat and Shen's latest protocol [41] implements a witness-indistinguishable proof that uses only symmetric key cryptographic operations. After the evaluator sends the blinded output of computation to the generator, the proof guarantees to the generator that the output value he received was actually generated by one of the garbled circuits he generated. However, it keeps the index of the circuit that produced the output hidden, as this could leak information to the generator.

3.4 Security Model and Definition

Our definition of security is based on the definition proposed by Kamara et al. [23], which we specify for the two-party setting as in Carter et al. [8]. We provide a brief description of the real/ideal world model here and direct readers to the previous work in this space for a more formal definition.

In the real world, both parties participating in the computation (the mobile device and application server) provide an input to the computation and an auxiliary input of random coins, while the single party designated as the outsourcing party (the Cloud) provides only random auxiliary input. The party evaluating the circuit in this computation is assumed to be non-colluding with the outsourcing party, as defined by Kamara et al. Some subset of these parties $A = (A_1, A_2, A_3)$ are corrupted and can deviate arbitrarily from the protocol. For the i^{th} honest party, let OUT_i be its output, and for the i^{th} corrupted party, let OUT_i be its view of the protocol

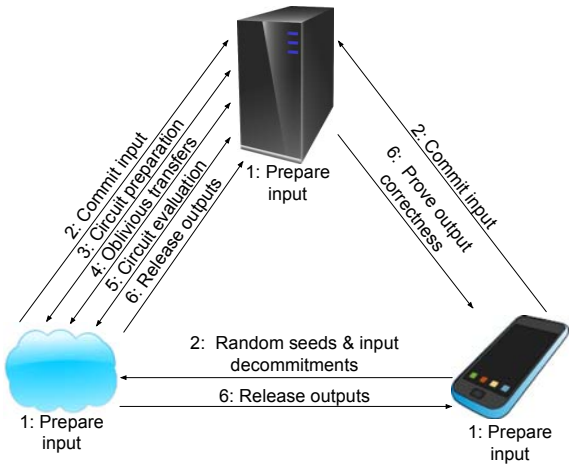


Figure 1: The complete Whitewash protocol. Note that MOBILE performs very little work compared to SERVER and CLOUD.

execution. Then the i^{th} partial output of a real protocol execution with input x is defined as:

$$REAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \| OUT_i$$

Where H is the set of honest parties, r is all random coins of all participants, and k is the security parameter.

In the ideal world, each party provides the same inputs as in the real world, however, they are sent to a trusted oracle which performs the secure computation. Once the trusted oracle completes the computation, it returns the output to the participating parties and no output to the outsourcing party. If any party aborts early or sends no input to the oracle, the oracle aborts and does not send the output to any party. For the i^{th} honest party, let OUT_i be its output to the oracle, and for the i^{th} corrupted party, let OUT_i be an arbitrary output value produced by the party. Then the i^{th} partial output of an ideal protocol execution in the presence of independent malicious simulators $S = (S_1, S_2, S_3)$ is defined as:

$$IDEAL^{(i)}(k, x; r) = \{OUT_j : j \in H\} \| OUT_i$$

Where H, r, and k are defined as before. Given this model, security is formally defined as:

DEFINITION 1. *An outsourcing protocol securely computes the function f if there exists a set of probabilistic polynomial-time (PPT) simulators $\{Sim_i\}_{i \in [3]}$ such that for all PPT adversaries (A_1, A_2, A_3), inputs x, auxiliary inputs z, and for all $i \in [3]$:*

$$\{REAL^{(i)}(k, x; r)\}_{k \in N} \overset{c}{\equiv} \{IDEAL^{(i)}(k, x; r)\}_{k \in N}$$

Where $S = (S_1, S_2, S_3)$, $S_i = Sim_i(A_i)$, and r is uniformly random.

4. PROTOCOL

4.1 Participants

Given a mobile device and a web or application server who wish to jointly compute a function, there are three participating parties in the Whitewash protocol:

- SERVER: "SERVER" refers to the web or application server participating in the joint computation. She is assumed to have large computational resources and is responsible for evaluating the garbled circuits.
- MOBILE: "MOBILE" refers to the mobile device participating in the joint computation. He is assumed to have limited processing power, memory, and communication bandwidth. MOBILE is tasked with garbling the circuit to be evaluated by SERVER.
- CLOUD: The outsourcing party "CLOUD" is responsible for relieving MOBILE of the majority of his computational load, but is not trusted with knowing either party's input to or output from the joint computation.

4.2 Protocol

Common Inputs: Security parameters k (key length) and σ (the number of circuits generated for the cut-&-choose); a commitment scheme $com(x; c)$ with committed value x and commitment key c; and a function $f(x, y)$.
Private Inputs: MOBILE inputs x and SERVER inputs y.
Outputs: Two outputs f_s, f_m for SERVER and MOBILE, respectively.

Phase 1: Pre-computation

1. Preparing inputs: MOBILE randomly generates $r \in \{0,1\}^{2k+\log(k)}$ as his input to the 2-universal circuit. He also generates $e \in \{0,1\}^{|f_m|}$ as a one-time pad for his output. SERVER computes her k-probe-resistant matrix M and $\bar{y}$ such that $M \cdot \bar{y} = y$. MOBILE'S input to the circuit will be $\bar{x} = x \| e \| r$ and SERVER'S input will be $\bar{y}$. We denote the set of indices $[m_s] = \{1, \cdots, |\bar{y}|\}$ and $[m_m] = \{1, \cdots, |\bar{x}|\}$.
2. Preparing circuit randomness: MOBILE generates random seeds $\{\sigma^{(j)}\}_{j \in [\sigma]}$ for generating the circuits and sends them to CLOUD.

Phase 2: Input commitments

1. Committing to MOBILE'S inputs: For each circuit $j \in [\sigma]$, input bit $i \in [m_m]$, and $b \in \{0,1\}$ MOBILE uses $\sigma^{(j)}$ to generate commitment keys $\nu_{i,b}^{(j)}$. Using the same random seeds, these keys will later be generated by CLOUD to commit to the input wire labels corresponding to MOBILE'S input. MOBILE then commits to his own inputs as $\{\Gamma^{(j)}\}_{j \in [\sigma]}$ as:

$$\Gamma^{(j)} = \{com(\nu_{i,\bar{x}_i}^{(j)}; \gamma_i^{(j)})\}_{i \in [m_m]}$$

using independently generated random commitment keys $\gamma_i^{(j)}$. MOBILE sends $\{\Gamma^{(j)}\}_{j \in [\sigma]}$ to SERVER and the commitment keys $\{\gamma_i^{(j)}\}_{i \in [m_m], j \in [\sigma]}$ to CLOUD.
2. Committing to CLOUD'S inputs: To allow for a fair release of the outputs, CLOUD inputs one-time pads to blind both parties' outputs. CLOUD randomly generates $p_s \in \{0,1\}^{|f_s|}$ and $p_m \in \{0,1\}^{|f_m|}$, as well as $r_c \in \{0,1\}^{2k+\log(k)}$ as its input to the 2-universal circuit. We denote CLOUD'S input as $z = p_s \| p_m \| r_c$, and the indices of CLOUD'S input wires as $[m_c] = \{1, \cdots, |z|\}$.
For each circuit $j \in [\sigma]$ and input bit $i \in [m_c]$, CLOUD uses $\{\sigma^{(j)}\}_{j \in [\sigma]}$ to generates the garbled input wire keys $(K_{i,0}^{(j)}, K_{i,1}^{(j)}, \pi_i^{(j)})$, where $K_{i,0}^{(j)}, K_{i,1}^{(j)} \in \{0,1\}^k$ and the permutation bit $\pi_i^{(j)} \in \{0,1\}$. To locate the correct key for bit b on input wire w_i of circuit j, we designate the label $W_{i,b}^{(j)} = (K_{i,b}^{(j)}, b \oplus \pi_i^{(j)})$.
Let $\{w_{m_s+i}\}_{i \in [m_c]}$ be the input wires for CLOUD. CLOUD then commits to the label pairs for its input wires as $\{\quad^{(j)}\}_{j \in [\sigma]}$,

where

$$\Psi^{(j)} = \{com(W^{(j)}_{m_s+i,0\oplus\pi_i^{(j)}}; \psi^{(j)}_{i,0\oplus\pi_i^{(j)}}), com(W^{(j)}_{m_s+i,1\oplus\pi_i^{(j)}}; \psi^{(j)}_{i,1\oplus\pi_i^{(j)}})\}_{i\in[m_c]}$$

using commitment keys $\psi^{(j)}_{i,b}$ generated with the random seed $\rho^{(j)}$. CLOUD then commits to its inputs as $\{\Xi^{(j)}\}_{j\in[\sigma]}$ as:

$$\Xi^{(j)} = \{com(\psi^{(j)}_{i,z_i}; \xi^{(j)}_i)\}_{i\in[m_c]}$$

using independently generated random commitment keys. CLOUD sends $\{\Psi^{(j)}\}_{j\in[\sigma]}$ and $\{\Xi^{(j)}\}_{j\in[\sigma]}$ to SERVER.

Phase 3: Circuit construction

1. **Constructing the objective circuit:** SERVER sends $\mathbf{M}$ to CLOUD, then SERVER and CLOUD run a coin flipping protocol to randomly determine the 2-universal hash matrix $\mathbf{H} \in \{0,1\}^{k\times m_m}$. These two matrices can be used to generate the new circuit C that computes the function $g : (\overline{x}, \overline{y}) \rightarrow (\perp, (h_m, h_c, c_s, c_m))$, where $h_m = \mathbf{H}\cdot\overline{x}, h_c = \mathbf{H}\cdot z, g_m = f_s(x, \mathbf{M}\cdot\overline{y}), c_m = g_m \oplus e \oplus p_m, g_s = f_s(x, \mathbf{M}\cdot\overline{y})$, and $c_s = g_s \oplus p_s$. MOBILE will need the values $h_c \| c_m$ to recover his output. We denote the set of indices corresponding to these values as $O_m = \{1, \cdots, |h_c| + |c_m|\}$.
2. **Committing to input and output wire label pairs:** Using the same method as in Phase 2, CLOUD uses $\{\rho^{(j)}\}_{j\in[\sigma]}$ to generate the input wire keys for both SERVER and MOBILE'S input as well as the output wire keys for MOBILE'S output (these output keys must be committed for the witness indistinguishable proof of MOBILE'S output correctness). Let $\{w_i\}_{i\in[m_m]}$ be the input wires for MOBILE, $\{w_{m_m+i}\}_{i\in[m_s]}$ be the input wires for SERVER, and $\{w_i\}_{i\in O_m}$. CLOUD then commits to the label pairs in MOBILE'S input, SERVER'S input, and MOBILE'S output as $\{\Theta^{(j)}, \Omega^{(j)}, \Phi^{(j)}\}_{j\in[\sigma]}$, where

$$\Theta^{(j)} = \{com(W^{(j)}_{i,0\oplus\pi_i^{(j)}}; \theta^{(j)}_{i,0\oplus\pi_i^{(j)}}), com(W^{(j)}_{i,1\oplus\pi_i^{(j)}}; \theta^{(j)}_{i,1\oplus\pi_i^{(j)}})\}_{i\in[m_m]}$$

$$\Omega^{(j)} = \{com(W^{(j)}_{m_m+i,0}; \omega^{(j)}_i), com(W^{(j)}_{m_m+i,1}; \omega^{(j)}_i)\}_{i\in[m_s]}$$

$$\Phi^{(j)} = \{com(W^{(j)}_{i,0}; \phi^{(j)}_i), com(W^{(j)}_{i,1}; \phi^{(j)}_i)\}_{i\in O_m}$$

using commitment keys generated with the random seed $\rho^{(j)}$. CLOUD then sends these commitments to SERVER.

Phase 4: Oblivious transfers (OT)

1. **Oblivious transfers:** CLOUD and SERVER execute m_s input oblivious transfers and σ circuit oblivious transfers as follows:
   (a) **Input:** For each $i \in [m_s]$, both parties run a $\binom{2}{1}$-OT where CLOUD inputs
   $$\left(\{(W^{(j)}_{m_m+i,0}; \omega^{(j)}_i)\}_{j\in[\sigma]}, \{(W^{(j)}_{m_m+i,1}; \omega^{(j)}_i)\}_{j\in[\sigma]}\right)$$
   while SERVER inputs $\overline{y}_i$. Once SERVER receives all of her garbled input wire labels, she uses the decommitment keys obtained in the OTs to check the committed wire values in $\{\Omega^{(j)}\}_{j\in[\sigma]}$. If any of the labels received in the OT do not match the committed wire labels, SERVER terminates the protocol.
   (b) **Circuit:** SERVER selects a set of circuits to be evaluated $S \subset [\sigma]$ such that $|S| = \frac{2\sigma}{5}$, as in shelat and Shen's protocol [40]. She represents this set with a bit string $s \in \{0,1\}^\sigma$ such that the j^{th} bit $s_j = 1$ if $j \in S$ and $s_j = 0$ otherwise. SERVER and CLOUD perform σ $\binom{2}{1}$-OTs where, for every $j \in [\sigma]$, CLOUD inputs $(\rho^{(j)}, (\{\gamma^{(j)}_i\}_{i\in[m_m]} \| \{\Xi^{(j)}_i\}_{i\in[m_c]}))$, while SERVER inputs s_j. This allows SERVER to learn either the randomness used to generate the check circuits or MOBILE and CLOUD'S inputs for the evaluation circuits without CLOUD knowing which circuits are being checked or evaluated.

Protocol	SYM	GROUP	OT	CT
CMTB	$\|x\|$	$\frac{2\sigma}{5}(\|y\|+1)$	k	yes
Salus	$\frac{2\sigma}{5}(\|x\|+\|y\|+\|f(x,y)\|)$	-	-	yes
WW	$\sigma(\|x\|+\frac{2}{5}\|f_m(x,y)\|)$	-	-	no

Table 1: Operations required on the mobile device by three outsourcing protocols. Here, SYM is the symmetric cryptographic operations, GROUP is the group algebraic operations, OT is the oblivious transfers, and CT is whether the protocol requires a coin toss. Recall that k is the security parameter, σ is the number of circuits generated, x is the mobile device's input, and y is the application server's input.

Phase 5: Evaluation

1. **Circuit evaluation:** Using $\rho^{(j)}$, CLOUD garbles the objective circuit C as $G(C)^{(j)}$ for all $j \in [\sigma]$ and pipelines these circuits to SERVER using Huang's technique [16]. Depending on whether the circuit is a check circuit or an evaluation circuit, SERVER performs one of two actions:
   (a) **Check:** For each $j \in [\sigma]\backslash S$, SERVER checks to see if $\rho^{(j)}$ can correctly regenerate the committed wire values $\{\Theta^{(j)}, \Omega^{(j)}, \Phi^{(j)}, \Psi^{(j)}\}$ and the circuit $G(C)^{(j)}$.
   (b) **Evaluate:** For each $j \in S$, SERVER checks that she can correctly decommit MOBILE'S input by recovering half of $\Theta^{(j)}$ from the keys committed in $\Gamma^{(j)}$. She does the same for CLOUD'S input, recovering half of $\Psi^{(j)}$ from the keys committed in $\Xi^{(j)}$

   If any of the above checks fail, SERVER aborts the protocol. Otherwise, she evaluates the circuits $\{G(C)^{(j)}\}_{j\in[\sigma]\backslash S}$. Each circuit outputs the values $(h^{(j)}_m, h^{(j)}_c, c^{(j)}_s, c^{(j)}_m)$ for $j \in [\sigma]\backslash S$.
2. **Majority output selection and consistency check:** Let (h_m, h_c, c_s, c_m) be the output of the majority of the evaluated circuits. If no majority value exists, SERVER aborts the protocol. Otherwise, she checks that $h^{(j)}_m = h_m$ and $h^{(j)}_c = h_c$ for all $j \in [\sigma]\backslash S$. If any of MOBILE or CLOUD'S hashed input values do not match, SERVER aborts the protocol.

Phase 6: Output proof and release

1. **Proof of output authenticity:** SERVER and MOBILE perform the proof of output authenticity from shelat and Shen's protocol [41] using the commitments to MOBILE'S output wires $\{\Phi^{(j)}\}_{j\in[\sigma]\backslash S}$ and the values $h_c \| c_m$.
2. **Output release:** CLOUD simultaneously releases the input one-time pads p_s and p_m to SERVER and MOBILE. SERVER and MOBILE then hash the pads and check to see if the hash values output by the circuit $h_c = H \cdot p_s \| p_m$. If the hashes do not match, SERVER and MOBILE abort the protocol. Otherwise, SERVER receives $c_s \oplus p_s$ as her output and MOBILE receives $c_m \oplus p_m \oplus e$ as his output.

5. COMPARISON WITH PREVIOUS OUTSOURCING PROTOCOLS

In this section, we compare the asymptotic complexity and security guarantees of the Whitewash protocol to two previous outsourcing techniques: the protocol developed by Carter et al. [8], which we call "CMTB" for the remainder of this work, and the Salus framework developed by Kamara et al. [23]. We refer to our

Whitewash protocol as WW.

When examining the complexity of each protocol, recall that one of our main goals is to optimize the efficiency on the mobile device. Thus, we examine the number of operations each protocol requires on the mobile device itself. When compared to the underlying shelat-Shen protocol, Whitewash adds extra input values from the Cloud, but does not add any steps to the computation that increase the complexity of operations performed on the application server or the Cloud. Thus, for a discussion of the application server and Cloud protocol complexity, we refer the reader to the original work by shelat and Shen [41].

5.1 Comparison to CMTB

The underlying two-party computation protocols of Whitewash and CMTB follow similar structures in terms of the security checks that are performed. However, Kreuter, shelat, and Shen's (KSS) protocol [28], which underlies CMTB, uses a number of algebraic operations to perform input consistency checks and output proofs of consistency. The protocol developed by shelat and Shen [41], which underlies Whitewash, removes these expensive cryptographic primitives in favor of constructions that use only efficient, symmetric-key operations. In addition to the improvements to the underlying protocol, Whitewash outsources the generation side of two-party computation, while CMTB outsources the evaluation side. In CMTB, since neither the mobile device or the Cloud could garble inputs before computation, a specially designed Outsourced Oblivious Transfer (OOT) protocol is necessary to deliver the mobile device's inputs to the evaluating Cloud in a secure, privacy-preserving manner. By swapping roles in the Whitewash protocol, we allow the mobile device to garble its own inputs, removing the need for an OT protocol to be performed from the mobile device. While Whitewash still requires OTs between the Cloud and the evaluating party, these operations can be parallelized, while the OOT protocol acts as a non-parallelizable bottleneck in computation.

5.1.1 Asymptotic Complexity

Table 1 shows this complexity for both Whitewash and CMTB. Note that for the mobile device, Whitewash requires significantly more symmetric key operations for garbling its own input and verifying the correctness of its output. By contrast, the OOT protocol in CMTB requires very few symmetric key operations, but requires several instantiations of an oblivious transfer. In addition, CMTB requires that the mobile device check the application server's input consistency and verify the correctness of the output using algebraic operations (e.g., modular exponentiations and homomorphic operations). Considering the fact that modular exponentiation is significantly more costly than symmetric key operations, removing these public key operations from the phone is a significant efficiency improvement for Whitewash. We also note that CMTB requires a two-party fair coin toss at the mobile device, which is not required by Whitewash.

5.1.2 Security Guarantees

The removal of the OOT protocol in Whitewash not only increases its efficiency when compared to CMTB, it also allows for stronger security guarantees. In CMTB, security was only possible if none of the parties collude, since the mobile device possessed information that would allow the Cloud to recover both input wire labels for all of the mobile input wires after the OOT. If the mobile device and Cloud collude in the Whitewash protocol, it simply removes the guarantee of fair release and makes the protocol equivalent to the underlying two-party computation protocol. Thus, the only guarantee lost is that of fair release at the end of the protocol, since a colluding mobile device and Cloud may not release the one-time pad used to blind the evaluating party's output. We believe that this represents a more realistic security setting, since the mobile device is paying for the assistance of the Cloud and may collude.

5.2 Comparison to Salus

When considering the operations performed on the mobile device, the Salus protocol and the Whitewash protocol both make the mobile device responsible for generating circuit randomness and garbling its own inputs. However, the Whitewash protocol requires an added proof of output consistency that is not included in Salus. While this proof adds some complexity to the protocol, it allows Whitewash to handle functions where both parties get different output values, while Salus is designed to handle functions with a single, shared output value. In addition, the Whitewash protocol outsources the generation of the garbled circuit, while the malicious secure Salus protocol outsources the evaluation. By swapping the roles of the outsourced task and adding in consistency checks at the evaluating party, the Whitewash protocol guarantees security in a stronger adversarial model.

5.2.1 Asymptotic Complexity

Both the Whitewash and Salus protocols use only efficient, symmetric key operations, but there is a slight tradeoff in the number of operations required (Table 1). Salus only requires operations for the $\frac{2\sigma}{5}$ evaluated circuits, but requires those operations for each bit of both party's inputs and the shared output. By contrast, Whitewash requires that the mobile device's input be committed for all σ circuits generated, but then only requires correctness proof of the output wires on the $\frac{2\sigma}{5}$ evaluated circuits. When the application server's input is significantly longer than the mobile device's, this will cause the Salus protocol to be less efficient than Whitewash. However, in the average case where both inputs are approximately the same length, this will mean that Whitewash requires more operations. This small tradeoff in efficiency is justified by the fact that Whitewash provides security in a stronger adversarial model than Salus. We also note that Salus requires a two-party fair coin toss before the protocol begins, which is not required by Whitewash.

5.2.2 Security Guarantees

The Salus protocol provides equivalent security guarantees to CMTB, guaranteeing security when none of the parties are colluding. This is a result of outsourcing the evaluation to the Cloud while allowing the mobile device to generate circuit randomness. If the mobile device colludes with the Cloud, they can trivially recover all of the other party's inputs. By outsourcing the generation of the garbled circuit and adding in additional consistency checks at the evaluating party, Whitewash guarantees security under this type of collusion. As stated above, the only guarantee lost is that of fair output release, which ultimately reduces Whitewash to the security of the underlying two-party computation protocol.

6. PERFORMANCE EVALUATION

Our protocol significantly expands upon the implementations of the PCF garbled circuit generation technique [27] and shelat and Shen's garbled circuit evaluation protocol [41]. For experimental comparison to previous protocols, we acquired the code implementation of the outsourcing protocol by Carter et al. [8] directly from the authors, as well as an Android port of the two-party garbled circuit protocol developed by Kreuter, shelat, and Shen [28]. We would like to thank the authors of [8, 27, 28, 41] for making their

Circuit	Input Size (Bits)	Total Gates		Non-XOR Gates	
		KSS	PCF	KSS	PCF
HAM (1600)	1,600	24,379	32,912	17,234	6,375
HAM (16384)	16,384	262,771	376,176	186,326	101,083
MAT (3x3)	288	424,748	92,961	263,511	27,369
MAT (5x5)	800	1,968,452	433,475	1,221,475	127,225
MAT (8x8)	2,048	8,067,458	1,782,656	5,006,656	522,304
MAT (16x16)	8,192	64,570,969	14,308,864	40,076,631	4,186,368
DIJK 10	112/1,040	259,232	530,354	118,357	291,490
DIJK 20	192/2,080	1,653,380	2,171,088	757,197	1,192,704
DIJK 50	432/5,200	22,109,330	13,741,514	10,170,407	7,549,370
RSA-256	256/512	934,092,960	673,105,990	602,006,981	235,925,023

Table 2: Input size and circuit size for all test circuits evaluated.

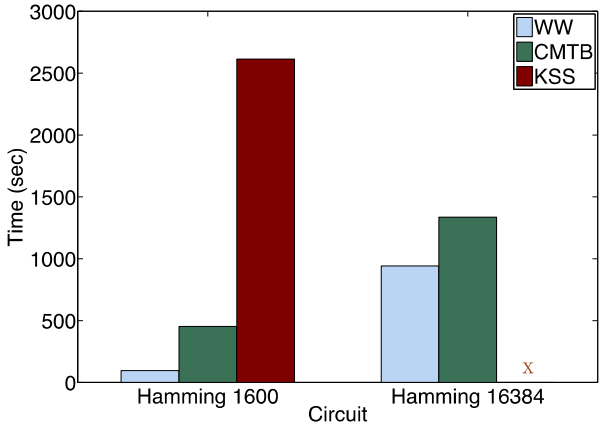


Figure 2: Execution time (ms) for Hamming Distance with input sizes of 1,600 and 16,384 bits for $\sigma = 256$ (note: log scale). Note that without outsourcing, only very small inputs can be computed over. Additionally, even for a large number of input bits, performing OTs on the servers still produces a faster execution time.

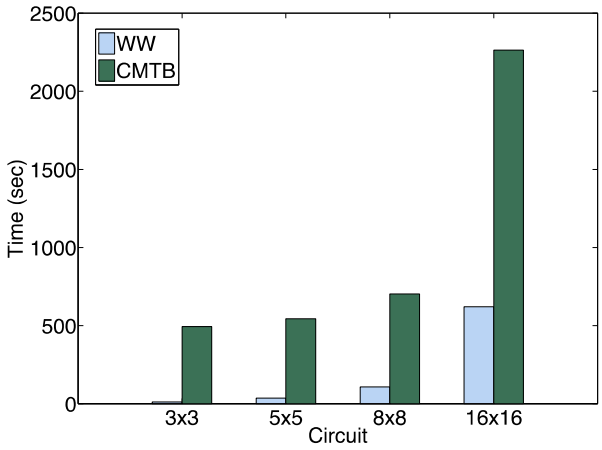


Figure 3: Execution time (ms) for the Matrix-Multiplication problem with input size varying between 3×3 matrices and 16×16 matrices for $\sigma = 256$ (note: log scale). This figure clearly shows that the oblivious transfers, consistency checks, and larger circuit representations of CMTB add up to a significant overhead as input size and gate count increase. By contrast, Whitewash requires less overhead and increases more slowly in execution time as gate counts and input size grow.

code available and for assisting us with our evaluation[2].

6.1 Test Environment

For evaluating our test circuits, we perform our experiments with a single server performing the role of Cloud and Application server, communicating with a mobile device over an 802.11g wireless connection. The server is equipped with 64 cores and 1TB of memory, and we partition the work between cores into parallel processing nodes using MPI. The mobile device used is a Samsung Galaxy Nexus with a 1.2 GHz dual-core ARM Cortex-A9 processor and 1 GB of RAM, running Android 4.0.

The large input sizes examined in the Hamming Distance trials required us to use a different testbed. For inputs as large as 16,384 bits, the phone provided by the above computing facility would overheat and fail to complete computation. Because the gate counts for Hamming Distance are significantly smaller than the other test circuits, we were able to run these experiments on a local testbed. We used two servers with Dual Intel Xeon E5620 processors, each with 4 hyper-threaded cores at 2.4 GHz each for the Cloud and the Application server. Each server is running the Linux kernel version 2.6, and is connected by a VLAN through a 1 Gbps switch. Our mobile device is a Samsung Galaxy Note II with a 1.6 GHz quad-core processor with Arm Cortex A9 cores and 2 GB of RAM, running the Android operating system at version 4.1. The phone connects to the two servers through a Linksys 802.11g wireless router with a maximum data rate of 54 Mbps. While this test environment represents optimistic connection speeds that may not always be available in practice, it allows us to consider the performance of the protocol without interference from variable network conditions, and mirrors the test environments used in previous work [8, 28, 41]. For all experiments except RSA-256, we take the average execution time over ten test runs, with a confidence interval of 95%. For RSA-256, we ran 3 executions.

6.2 Experimental Circuits

To evaluate the performance of our protocol, we run tests over the following functions. We selected the following test circuits because they exercise a range of the two major variables that affect the speed of garbled circuit protocols: input size and gate counts.

[2]We contacted the authors of the Salus protocol [23] in an attempt to acquire their framework to compare the actual performance of their scheme with ours. Because they were unable to release their code, no sound comparison to their work beyond an asymptotic analysis was possible. Our code is available at `http://www.foryourphoneonly.org`.

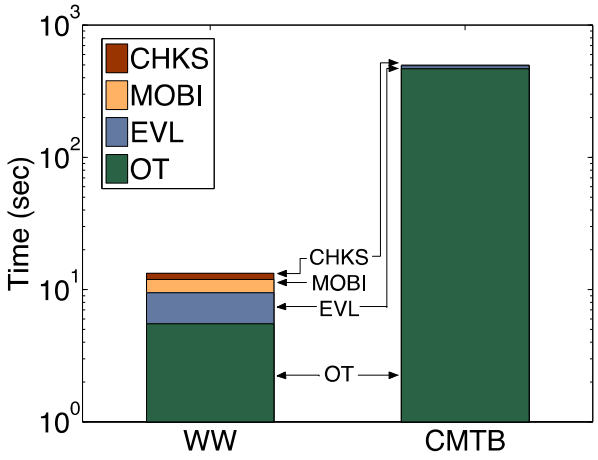


Figure 4: Microbenchmarking execution times (ms) for Whitewash and CMTB over the Matrix-Multiplication problem. We denote the total time spent in computation for Whitewash as "MOBI". Since the mobile device is linked with "CHKS" and "OT" in CMTB, we do not separate out the mobile time for that protocol. Notice the dominating amount of time required to perform oblivious transfers. Moving these operations off the mobile device removes a significant computation bottleneck.

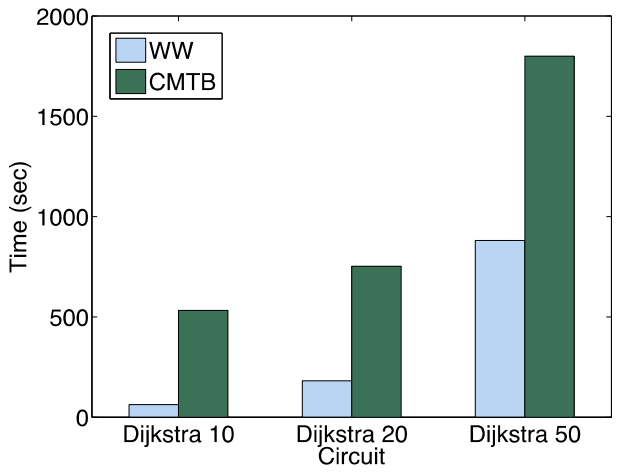


Figure 5: Execution time (s) for Dijkstra's algorithm with input sizes of $10, 20,$ and 50 node graphs for $\sigma = 256$. This figure shows that the Whitewash protocol allows for computation that was only feasible to be executed in a close to practically useful time frame.

In addition, these programs are becoming somewhat standard test applications, having been used as benchmarks in a large amount of the related literature [8, 27, 28, 41]. All of the programs are implemented with the algorithms used by Kreuter et al. [27] except for Dijkstra's algorithm, which matches the implementation used by Carter et al. [8]:

- **Hamming Distance (HAM):** The Hamming Distance circuit accepts binary string inputs from both parties and outputs the number of locations at which those strings differ. This circuit demonstrates performance for a small number of gates over a wide range of input sizes. We consider input strings of length 1,600 bits and 16,384 bits.
- **Matrix Multiplication (MAT):** Matrix multiplication takes an $n \times n$ matrix of 32-bit integer entries from each party and outputs the result of multiplying the matrices together. This circuit demonstrates performance when both input size and gate count vary widely. We consider square matrix inputs where $n = 3, 5, 8,$ and 16.
- **Dijkstra's Algorithm (DIJK):** This version of Dijkstra's algorithm takes an undirected weighted graph with a grid structure and a maximum node degree of four from the first party, and a start and end node from the second party. The circuit outputs the shortest path from the start node to the end node to the second party, and nothing to the first. For an n node graph, the graph description from the first party requires $104n$ input bits, while the start and end node descriptions require $8n + 32$ bits. We consider graphs with $n = 10, 20,$ and 50 nodes. Due to an error in the PCF compiler, we were unable to compile a program for graphs larger than 50 nodes.
- **RSA Function (RSA):** The RSA function (i.e., modular exponentiation) accepts an RSA message from one party and an RSA public key from the other party and outputs the encryption of the input plaintext under the input public key. Specifically, one party inputs the modulus $n = pq$ for primes p and q, as well as the encryption key $e \in \mathbb{Z}_{\phi(n)}$. The other party inputs a message $x \in \mathbb{Z}_n^*$, and the circuit computes $x^e \pmod n$. This circuit demonstrates performance for small input sizes over very large gate counts. We consider the case where the input values $x, n,$ and e are 256 bits each. While these are not secure parameters in practice, the function itself provides a complex circuit that is scalable on input size and useful for benchmarking our protocol.

For each test circuit, we consider the time required to execute and the bandwidth overhead to the mobile device. Table 2 shows the input size and gate counts for each test circuit, showing the exact range of values tested for these two circuit variables.

6.3 Execution Time

In all experiments, the efficiency gains of removing oblivious transfers and public key operations are immediately apparent. To examine how Whitewash compares to generating garbled circuits directly on the mobile device, we considered Hamming Distance as a simple problem (Figure 2). Even with a relatively small gate count, garbling the circuit directly on the mobile device is only possible for the small input size of 1,600 bits. Whitewash is capable of executing this protocol in 96 seconds, while running the computation directly on the mobile device takes 2,613 seconds, representing a 96% performance improvement through our outsourcing scheme. For the very large input size of 16,384 bits, computation directly on the mobile device ceases to be possible. When comparing to CMTB, this circuit further illustrates the cost of oblivious transfers on the mobile device. Even with the significantly reduced number of OTs allowed by the OOT protocol in CMTB (80 OTs), performing 16,384 malicious secure oblivious transfers between two servers in Whitewash still runs 30% faster than CMTB.

The matrix-multiplication circuit provides a good overview of average-case garbled circuit performance, as it represents a large range of both gate counts and size of inputs. For the input size of a 3×3 matrix, the Whitewash protocol runs in an average of 12 seconds, while CMTB requires 493 seconds, representing a 98% improvement (see Figure 3). Upon inspecting the micro benchmarking breakdown of each protocol's execution in Figure 4, we observe a significant speedup simply by moving oblivious transfers off of the mobile device. Even though the number of OTs required by CMTB is essentially constant based on their application of the

Ishai OT extension, performing standard malicious secure oblivious transfers in parallel between the servers is much more efficient than requiring that the phone perform these costly operations. In addition, if we examine the amount of execution time where the phone participates in Whitewash, we see that the mobile device ("MOBI" in Figure 4), takes around 1 second, and is idle during the majority of computation. By contrast, both the OT and consistency check phases of CMTB require the mobile device to participate in a significant capacity, totaling to almost 8 minutes of the computation. Having the phone perform as little work as possible means that the Whitewash protocol performance is nearly equivalent to running the same computation between two server-class machines.

To examine the performance of Whitewash for a more practical application, we considered the Dijkstra's algorithm circuit used to implement privacy-preserving navigation by Carter et al. [8]. They point out that this application, which has uses from military convoys to industrial shipping routes, is a significant first step in providing privacy for the growing genre of location-based mobile applications. Unfortunately, the PCF compiler does not optimize the Dijkstra's circuit as well as the previous experimental programs, which is evident in Table 2. In the 10 and 20 node graphs, the PCF compiler even produces a *larger* circuit than the compiler used by KSS. However, despite evaluating larger circuits, the Whitewash protocol still outperforms CMTB in execution time, running 88%,76%, and 51% faster in the 10, 20, and 50 node cases respectively (shown in Figure 5). As circuit compilers continue to improve and produce smaller circuits, the performance gains of the Whitewash protocol will be even larger. In this experiment, we also noticed that because Whitewash evaluates and checks circuits simultaneously, it created contention for the network stack in our test server. In a truly distributed environment where each server node has dedicated network resources, the highly parallelizable structure of shelat and Shen's protocol would allow Whitewash to execute faster. Given that Whitewash can execute Dijkstra's algorithm obliviously on the order of minutes, it allows computation considered only feasible for previous schemes to be performed in a nearly practical execution time.

The previous experiments clearly show that outsourcing is necessary to run circuits of any practical size. For our final test circuit, we consider an extremely complex problem to demonstrate the ability of outsourcing protocols in the worst-case. The RSA-256 circuit evaluated by Kreuter et al. in [27] and shelat and Shen in [41] represents one of the largest garbled circuits ever evaluated by a malicious secure protocol. For the RSA-256 problem, Whitewash completed the computation in 515 minutes. CMTB was unable to complete one execution of the protocol. A large part of this efficiency improvement results from the underlying protocol of Whitewash, which uses only symmetric-key operations outside of the oblivious transfers between the servers. The reduced non-XOR gate counts and more compact circuit representation of the PCF compiler also contribute to this improvement. Ultimately, because Whitewash ensures that the phone participates minimally in the protocol, it no longer acts as a bottleneck on computation. We essentially reduce performance of our outsourcing protocol to that of the underlying two-party protocol, allowing this technique for outsourcing to benefit as more improvements are made in non-outsourced garbled circuit protocols. In addition, this minimal level of interactivity allows us to run these protocols with 256 circuits, equivalent to a security parameter of approximately 80-bit security, which is agreed by the research community to be an adequate security parameter. Finally, the phone is only active for a few seconds during this large computation, keeping its system resources free for other user applications (or to preserve battery power) while the servers complete the computation. *This shows that Whitewash is capable of evaluating the same circuits as the most efficient desktop-based garbled circuit protocols with a minimal overhead cost*. Exact execution times are shown in our technical report [7].

	Bandwidth (MB)			Reduction Over	
Circuit	WW	CMTB	KSS	CMTB	KSS
HAM (1600)	23.56	41.05	240.33	42.62%	90.20%
HAM (16384)	241.02	374.03	x	35.56%	x
MAT (3x3)	4.26	11.50	x	62.97%	x
MAT (5x5)	11.79	23.04	x	48.82%	x
MAT (8x8)	30.15	51.14	x	41.05%	x
MAT (16x16)	120.52	189.52	x	36.41%	x
DIJK 10	1.67	20.21	x	91.73%	x
DIJK 20	2.85	35.28	x	91.93%	x
DIJK 50	6.38	80.49	x	92.08%	x
RSA-256	3.97	x	x	x	x

Table 3: Bandwidth measures for all experiment circuits. Note that there is as much as a 84% reduction in bandwidth when using the Whitewash protocol.

6.4 Network Bandwidth

The Whitewash protocol not only improves the speed of execution when outsourcing garbled circuit computation, it also significantly reduces the amount of bandwidth required by the mobile device to participate in the computation. Table 3 shows the bandwidth used by the mobile device for each test circuit. In the best case, for Dijkstra's algorithm over 50 node graphs, we observed a 92% reduction in bandwidth between Whitewash and CMTB. This is a result of the mobile device not performing OTs and only sending relatively small symmetric-key values instead of algebraic elements for consistency checks. For all test circuits, we observed a small decrease in the amount of improvement between the two protocols as the input size increased. This is because the number of commitments sent by the phone in Whitewash increases as the size of the input grows, while CMTB performs a fixed number of OTs as the input size increases. However, the oblivious transfers still require a significant enough amount of bandwidth to make removing them the most efficient option. When comparing to not outsourcing garbled circuit generation, the cost of oblivious transfers and sending several copies of the garbled circuit to the evaluator quickly adds up to a significant bandwidth cost. For the smallest circuit evaluated, outsourcing the circuit garbling reduces the required amount of bandwidth by 90%. The importance of these bandwidth reductions is further highlighted when considering mobile power savings. With data transmission costing roughly 100 times as much power as computation on the same amount of data, any reduction in the bandwidth required by a protocol implies a critical improvement in practicality.

One challenge encountered during the implementation of the Whitewash protocol was the extensive use of hardware-specific functions used to implement commitment schemes in shelat and Shen's code. Rather than try to port this code over to Android, which would require significant development of hardware-specific libraries, we chose to implement the protocol in an equivalent manner by having the Cloud generate the part of the commitments which requires these functions and send them to the mobile device. The mobile device then finishes generating the commitments that match its input and forwards them to the evaluator. Our proofs of security remain valid even with this small protocol modification. Our preliminary implementation using instructions specific to the ARM architecture has shown that we could further reduce the measured bandwidth values by over 60%. With already significant bandwidth

reductions from previous outsourcing schemes, our protocol will see further improvements as mobile hardware begins to incorporate more machine-specific libraries.

7. CONCLUSION

With the increasingly pervasive and personal nature of mobile computing, garbled circuits provide a solution that preserves both privacy and application functionality. However, to make these computationally expensive protocols usable on mobile devices, secure outsourcing to the cloud is necessary. We develop a new scheme that eliminates the most costly operations, including oblivious transfers, from the mobile device. By requiring that the mobile device instead produce the randomness required for circuit generation, we significantly reduce the number of algebraic group operations and communication rounds for the mobile device. At the same time, we bolster the security guarantees against certain types of collusion, yielding a more secure protocol than any other in this space. Our performance evaluation shows average gains of 75% for execution time and 60% for bandwidth over the previous outsourcing protocol. These improvements allow large circuits representing practical applications to be computed efficiently from a mobile device. As a result, we show that garbled circuit protocols can be made nearly as efficient for mobile devices as they are for server-class machines.

Acknowledgments This material is based on research sponsored by DARPA under agreement number FA8750-11-2-0211. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] M. J. Atallah and K. B. Frikken. Securely outsourcing linear algebra computations. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.

[2] Y. Aumann. Security Against Covert Adversaries: Efficient Protocols for Realistic Adversaries. *Journal of Cryptology*, 18(3):554–343, 2010.

[3] D. Beaver. Server-assisted cryptography. In *Proceedings of the workshop on New security paradigms (NSPW)*, 1998.

[4] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient garbling from a fixed-key blockcipher. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2013.

[5] J. Brickell and V. Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *Proceedings of the international conference on Theory and Application of Cryptology and Information Security*, 2005.

[6] H. Carter, C. Amrutkar, I. Dacosta, and P. Traynor. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks (SCN)*, 7(7):1165–1176, 2014.

[7] H. Carter, C. Lever, and P. Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. Cryptology ePrint Archive, Report 2014/224, 2014. `http://eprint.iacr.org/`.

[8] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of the USENIX Security Symposium*, 2013.

[9] comScore. comScore Reports February 2013 U.S. Smartphone Subscriber Market Share. `http://www.comscore.com/Insights/Press_Releases/2013/4/comScore_Reports_February_2013_U.S._Smartphone_Subscriber_Market_Share`, 2013.

[10] I. Damgård, M. Geisler, and J. B. Nielsen. From passive to covert security at low cost. In *Proceedings of the 7th international conference on Theory of Cryptography*, 2010.

[11] I. Damgard, V. Pastro, N. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2012.

[12] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology - CRYPTO*, 2012.

[13] S. D. Gordon, J. Katz, V. Kolesnikov, A.-l. B. Labs, F. Krell, and M. Raykova. Secure Two-Party Computation in Sublinear (Amortized) Time. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[14] M. Green, S. Hohenberger, and B. Waters. Outsourcing the Decryption of ABE Ciphertexts. In *Proceedings of the USENIX Security Symposium*, 2011.

[15] C. Hazay and Y. Lindell. Efficient Protocols for Set Intersection and Pattern Matching with Security Against Malicious and Covert Adversaries. *Journal of Cryptology*, 23(3):422–456, 2008.

[16] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[17] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[18] Y. Huang, J. Katz, and D. Evans. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology–CRYPTO*, 2013.

[19] N. Hustead, S. Myers, abhi shelat, and P. Grubbs. GPU and CPU parallelization of honest-but-curious secure two-party computation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2013.

[20] A. Iliev and S. W. Smith. Small, Stupid, and Scalable: Secure Computing with Faerieplay. In *The ACM Workshop on Scalable Trusted Computing*, 2010.

[21] S. Jha, L. Kruger, and V. Shmatikov. Towards practical privacy for genomic computation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[22] S. Kamara, P. Mohassel, and M. Raykova. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. `http://eprint.iacr.org/`.

[23] S. Kamara, P. Mohassel, and B. Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[24] F. Kerschbaum. Collusion-resistant outsourcing of private set intersection. In *Proceedings of the ACM Symposium on Applied Computing*, 2012.

[25] M. Kiraz and B. Schoenmakers. A Protocol Issue for The Malicious Case of Yao's Garbled Circuit Construction. In *Proceedings of the Symposium on Information Theory in the Benelux*, 2006.

[26] M. S. Kiraz. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[27] B. Kreuter, a. shelat, B. Mood, and K. Butler. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings of the USENIX Security Symposium*, 2013.

[28] B. Kreuter, a. shelat, and C. Shen. Billion-Gate Secure Computation with Malicious Adversaries. In *Proceedings of the USENIX Security Symposium*, 2012.

[29] L. Kruger, S. Jha, E.-J. Goh, and D. Boneh. Secure Function Evaluation with Ordered Binary Decision Diagrams. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2006.

[30] Y. Lindell. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology–CRYPTO*, 2013.

[31] Y. Lindell and B. Pinkas. Privacy preserving data mining. In *Proceedings of the Annual International Cryptology Conference on Advances in Cryptology*, 2000.

[32] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the annual international conference on Advances in Cryptology*, 2007.

[33] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the conference on Theory of cryptography*, 2011.

[34] L. Malka. Vmcrypt: modular software architecture for scalable secure computation. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[35] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–a secure two-party computation system. In *Proceedings of the USENIX Security Symposium*, 2004.

[36] A. Miyaji and M. S. Rahman. Privacy-preserving data mining in presence of covert adversaries. In *Proceedings of the international conference on Advanced data mining and applications: Part I*, 2010.

[37] P. Mohassel and M. Franklin. Efficiency tradeoffs for malicious two-party computation. In *Proceedings of the Public Key Cryptography conference*, 2006.

[38] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[39] N. Nipane, I. Dacosta, and P. Traynor. "Mix-In-Place" anonymous networking using secure function evaluation. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2011.

[40] a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *Proceedings of the Annual international conference on Theory and applications of cryptographic techniques*, 2011.

[41] a. shelat and C.-H. Shen. Fast two-party secure computation with minimal assumptions. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2013.

[42] D. Talbot. Security in the ether. `http://www.technologyreview.com/featuredstory/416804/security-in-the-ether/`, 2009.

[43] A. C. Yao. Protocols for secure computations. In *Proceedings of the Annual Symposium on Foundations of Computer Science*, 1982.

Outsourcing Secure Two-Party Computation as a Black Box

Abstract

Secure multiparty computation (SMC) offers a technique to preserve functionality and data privacy in mobile applications. Current protocols that make this costly cryptographic construction feasible on mobile devices securely outsource the bulk of the computation to a cloud provider. However, these outsourcing techniques are built on specific secure computation assumptions and tools, and applying new SMC ideas to the outsourced setting requires the protocols to be completely rebuilt and proven secure. In this work, we develop a generic technique for lifting any secure two-party computation protocol into an outsourced two-party SMC protocol. By augmenting the function being evaluated with auxiliary consistency checks and input values, we can create an outsourced protocol with low overhead cost. Our implementation and evaluation show that in the best case, our outsourcing additions execute within the confidence intervals of two servers running the same computation, and consume approximately the same bandwidth. In addition, the mobile device itself uses minimal bandwidth over a single round of communication. This work demonstrates that efficient outsourcing is possible with any underlying SMC scheme, and provides an outsourcing protocol that is efficient and directly applicable to current and future SMC techniques.

1 Introduction

As the mobile computing market continues to grow, an increasing number of mobile applications are requiring users to provide personal or context-sensitive information. However, as the recent iCloud breach demonstrates [25], these application servers cannot necessarily be trusted to maintain the security of the data they possess. To better preserve privacy and the functionality of mobile applications, secure multiparty computation (SMC) techniques offer protocols that allow application servers to process user data while it remains encrypted. Unfortunately, while a plethora of SMC techniques exist, they currently require too much processing power and device memory to be practical on the mobile platform. Furthermore, the bandwidth and power requirements for these SMC protocols will always be a limiting requirement for mobile applications even as the computational resources of mobile devices grow.

To bring SMC to the mobile platform in a more efficient way, recent work has focused on developing secure techniques for outsourcing the most expensive computation. Rather than naively trusting the Cloud to stand in for the mobile device in a standard SMC protocol, these outsourced protocols seek to use the Cloud for computation without revealing any input or output values. A number of these protocols have been specifically developed to outsource garbled circuit protocols [24, 8, 7]. These protocols attempt to optimize the outsourcing operations without increasing the complexity of the circuit being evaluated. However, because of this optimization goal, they are constructed and proven secure using specific garbled circuit evaluation techniques. As new techniques for SMC are developed that modify the garbled circuit construction (or use completely different underlying constructions), it is unclear whether these specific outsourcing protocols will be able to take advantage of the new developments.

In this work, we develop a technique for outsourcing secure two-party computation for *any* two-party SMC technique. Rather than avoiding changes to the function being evaluated, we add a small amount of overhead to the evaluated function itself. This tradeoff allows for an outsourcing scheme that relies on the underlying two-party protocol in a black-box manner, meaning the underlying protocol can be swapped for any other protocol meeting the same definition of security This makes the task of securely incorporating newly developed SMC techniques trivial. This protocol enables mobile devices to participate in *any secure two-party SMC protocol* with

minimal cost to the device and with nominal overhead to the servers running the computation. Specifically, we make the following contributions:

- **Develop a black-box outsourcing protocol:** We develop a novel outsourcing technique for lifting any two-party SMC protocol into the two-party outsourced setting. To do this, we add a small amount of overhead to the function being evaluated to ensure that none of the inputs are modified by malicious participants. This technique of augmenting the evaluated circuit has been successfully used in other SMC protocols to balance performance with security guarantees [19, 29, 41]. In addition, we leverage the non-collusion assumption used throughout the related work to produce an output consistency check that incurs trivial overhead. While this approach slightly increases the cost of evaluation, it minimizes the computation and bandwidth required by the mobile device.

- **Prove security for any underlying two-party SMC protocol:** We provide simulation proofs of security to demonstrate that our protocol is secure in the malicious threat model. The only requirement of the underlying two-party SMC protocol is that it satisfy the canonical ideal/real world simulation definition of security against malicious adversaries [14]. This allows *any* future SMC protocols that are developed to be used in a plug-&-play manner with our outsourcing technique.

- **Implement and evaluate the overhead cost of the outsourcing operations:** Using the garbled circuit two-party SMC protocol of shelat and Shen [41], we implement our protocol and evaluate the complete overhead cost of outsourcing. Rather than compare to previous outsourcing schemes, we instead measure the overhead incurred by augmenting the desired functionality, as well as the input and output preparation and checking. This measurement of cost better represents the value of the scheme, as a direct comparison to previous outsourcing protocols would drastically change depending on the underlying two-party SMC protocol implemented in our scheme. Our results show that for large circuits, black-box outsourcing incurs negligible overhead (i.e., the confidence intervals for outsourced and server only execution intersect) in evaluation time and in bandwidth required when compared to evaluating the unmodified function. To demonstrate the practical performance of our protocol, we develop a mobile-specific facial recognition application and analyze its performance.

The rest of this work is organized as follows: Section 2 describes related research, Section 3 outlines definitions of security, Section 4 formally defines the protocol, Section 5 provides an overview of security, Section 6 describes our implementation and performance evaluation, Section 7 presents a new mobile-specific application for SMC, Section 8 compares the overhead of our black box technique to previous work, and Section 9 provides concluding remarks.

2 Related Work

Since it was initially conceived in the early 1980's [42, 15], secure multiparty computation (SMC) has grown from a theoretical novelty to a potentially useful and practical cryptographic construction. The FairPlay implementation [33] provided one of the first schemes for performing secure multiparty computation in practice. Since then, a number of other protocols and implementations have shown that privacy-preserving computation in the semi-honest threat model can be performed relatively efficiently [18, 4, 1]. However, this security model is weak in practice, and does not provide enough security for most real-world situations. To resolve this, recent study has focused on developing protocols that are secure in the malicious setting. For two-party computation, the garbled circuit construction has seen a large amount of new development [30, 31, 34, 39, 28, 40, 41] that has drastically reduced the cost of circuit checking and the associated consistency verification. Because the cut-&-choose construction that is typically applied in this setting is very costly, recent work has sought to minimize the cost of the cut-&-choose [12, 29, 20] or amortize that cost over a batch of circuit executions [32, 21]. Besides the garbled circuit technique, other techniques using somewhat homomorphic encryption [10, 9] and oblivious transfer [37] have shown promise of producing efficient protocols for secure multiparty computation in the malicious threat model. However, all of these techniques still have significant overhead cost that makes them infeasible to execute without sizable computational resources.

With smartphone applications retrieving private user data at an increasing rate, secure multiparty computation could potentially offer a way to maintain privacy and functionality in mobile computing. However, the efficiency challenges of secure multiparty computation are compounded when considered in the resource-constrained mobile environment. Previous work has shown that smartphones are generally limited to simple functions in the semi-honest setting [6, 17]. Demmler et al. [11] showed how to incorporate pre-computation on hardware tokens to improve efficiency on mobile devices, but still in the semi-honest setting. In addition

to the cost of evaluating these SMC protocols, Mood et al. [36] and Kreuter et al. [27] demonstrated that even with significant optimization, the task of compiling circuits on the mobile device can also be quite costly.

Given these limitations, evaluating SMC protocols directly on mobile hardware does not seem to be possible in the immediate future. Because of this, mobile secure computation research has recently focused on applying techniques from server-assisted cryptography [3] to move the most costly cryptographic operations off of the mobile device and onto a more capable cloud server. To achieve this, many authors have focused on developing protocols for outsourcing secure computation of specific algorithms such as graph algorithms [5], set intersection [26], and linear algebra functions [2]. The first protocol to outsource secure multiparty computation for any function was developed by Kamara et al. [23, 24]. In this work, the authors established a definition of security that assumes specific parties in the computation, while malicious, are not allowed to collude. Following on this definition, several other protocols and efficiency improvements have been developed for the outsourced setting [8, 35, 7]. Unfortunately, all of these protocols are built on specific secure multiparty computation assumptions and techniques. With new and varying techniques for SMC being developed at a rapid pace, it is unclear how to apply the outsourcing techniques used in these protocols to new schemes to allow them to benefit from new efficiency improvements. In this work, we seek to develop a protocol that can lift *any* two-party SMC protocol into the outsourced setting with little overhead.

In concurrent work to our own, Jakobsen et al. [22] develop a framework for outsourcing secure computation that can be used to describe our protocol. However, our protocol uses techniques modified to fit the mobile application model. Furthermore, our work provides an implementation, an empirical performance analysis, and develops a mobile-specific application with a performance characterization.

3 Definitions of Security

Outsourced two-party SMC protocols are designed to allow two parties of asymmetric computational capability to engage in a privacy-preserving computation with the assistance of an outsourcing party. We consider the situation where a mobile device possessing limited computational resources wishes to run an SMC protocol with an application server or other well-provisioned entity. To allow this, outsourcing protocols move the majority of the costly operations off of the mobile device and onto a Cloud provider *without* revealing to the Cloud either party's input or output to the computation. These protocols aim to provide security guarantees of privacy and correctness, and also attempt to minimize the computation required at the mobile device while still maintaining efficiency between the application server and the Cloud. To meet these goals in the outsourced setting, a number of careful security assumptions must be made.

3.1 Two-party SMC security

Our black box protocol is based on the execution of a two-party SMC protocol to obliviously compute the result. We make no assumptions about the techniques used or structure of this underlying protocol except that it meets the canonical definition of security against malicious adversaries using the ideal/real world paradigm [14]. Informally, this states that for any adversary participating in the two-party SMC protocol, there exists a simulator in an ideal world with a trusted third party running the computation where the output in both worlds is computationally indistinguishable. In this definition, the simulator in the ideal world is given oracle access to the adversary in the real world. Particularly in the two-party setting, there are a few caveats that must be assumed to make this definition feasible, and must be considered when designing an outsourced protocol that uses a two-party protocol in a black box manner.

First, it is known that two-party protocols cannot fully prevent early termination. In any execution, one party will receive their output of computation before the other party does. While certain techniques have been developed to partially solve this problem, there is no complete solution. While other outsourcing protocols have added in a fair-release guarantee, this guarantee comes at a cost. Either the protocol must provide additional commitments not guaranteed in a standard two-party protocol [8, 7], or the protocol must incorporate additional costly MAC operations to ensure the output is not tampered with [24, 35]. However, our black box protocol shows that if we treat the outsourced model like a standard two-party execution where fair release is not guaranteed, we can reduce the output consistency check to a single comparison on the mobile device. This allows the application server to recover its input first and potentially disrupt the mobile device's output, but mirrors the two-party execution guarantees exactly. Thus, our protocol optimizes execution overhead by not assuming a fair output release.

Second, it is possible that a malicious party can provide arbitrary input to the computation that may or may not correspond to their "real" input. While we cannot control what another party provides as input to the computation, this potential behavior must be handled by the definition of security. To handle this, the simulator in the ideal world, which has oracle access to the adversary in the real world, must not only be able to simulate the ad-

versary's view of the protocol. Upon running the adversary with a given input, the simulator must also be able to recover the actual input used by the adversary. In our proofs of execution, the ideal world will invoke this simulator often as a mechanism to recover the adversary's input before initiating computation with the trusted third party. This ensures that the output in both worlds is indistinguishable.

Given these assumptions, a secure two-party SMC protocol provides two guarantees. The first is privacy, which means that a malicious adversary cannot learn anything about the other party's input or output value beyond what is revealed by his own output value. The second guarantee is correctness. This implies that even in the presence of a malicious adversary, the output of the protocol will be the correct output of the agreed upon function except with negligible probability.

For a formal definition of security and further discussion, refer to [14].

3.2 Collusion assumptions

Previous work in outsourcing secure multiparty computation makes careful assumptions about who in the computation is allowed to collude. Kamara et al. [24] discuss at length the theoretical justification for these assumptions. Essentially, to achieve an n-party outsourcing protocol with better complexity than a two-party SMC protocol, it must be assumed that the Cloud (i.e., the server aiding computation but not providing input to the function) cannot collude with any other party. Other outsourcing protocols have sought ways to relax this restriction without significantly increasing the complexity of the function being evaluated [8, 7]. However, all of these protocols still assume that the application server and the Cloud cannot collude. We follow this assumption in our black box construction. As stated by Kamara et al., the existence of an outsourcing protocol where this particular collusion is allowed would imply an efficient two-party SMC scheme where one party performs work that is sub-linear with respect to the size of the function being evaluated. While there are techniques for such a two-party SMC protocol [13, 16], it is unclear that they can be applied to create such an outsourced protocol.

3.3 Outsourced Security Definition

We follow the security definition first established by Kamara et al. [24] but specified for the two-party scenario as in the work of Carter et al. [8, 7]. We slightly alter the definition to allow for the possibility of early termination by one of the parties, possibly preventing the other party from receiving output. We provide a summary of the definition here, and refer the reader to previous work for a complete discussion of the definition.

The real world setting is made up of three parties. Two of these parties provide input to the computation, while the third party takes on computational load for one of the two input parties. All three parties provide auxiliary random inputs to the protocol. Some subset of the three parties $A = (A_1, A_2, A_3)$ can behave maliciously, but we assume that the application server and the Cloud cannot collude. For the i^{th} honest party, OUT_i is defined as its output, and for the i^{th} corrupted party, OUT_i is its view of the protocol. Then we define the i^{th} partial output as:

$$REAL^{(i)}(k,x;r) = \{OUT_j : j \in H\} \cup OUT_i$$

Here, k is the security parameter, x is all inputs to the computed function, r is the auxiliary randomness, and H is the set of all honest parties.

The ideal world setting is made up of the same parties with the same inputs as the real world with the addition of a trusted third party that receives all parties' inputs, computes the desired function, and returns the output to all parties except the outsourced party that is not providing inputs to the function. Any party may abort the computation early or refuse to send input, in which case the trusted party sends no output. As in the standard two-party definition [14], it is possible for one party, upon receiving output from the trusted third party, to terminate the protocol, preventing the other party from receiving its output. For the i^{th} honest party, OUT_i is defined as its output received from the trusted party, and for the i^{th} corrupted party, OUT_i is an arbitrary output value. Then we define the i^{th} partial output in the presence of independent malicious simulators $S = (S_1, S_2, S_3)$ as:

$$IDEAL^{(i)}(k,x;r) = \{OUT_j : j \in H\} \cup OUT_i$$

Here, k, x, r, and H are defined as above. In this real/ideal world setting, outsourced security is defined as follows:

Definition 1. *An outsourcing protocol securely computes the function f if there exists a set of probabilistic polynomial-time (PPT) simulators $\{Sim_1, Sim_2, Sim_3\}$ such that for all PPT adversaries (A_1, A_2, A_3), inputs x, and for all $i \in \{1,2,3\}$:*

$$\{REAL^{(i)}(k,x;r)\}_{k\in N} \overset{c}{\approx} \{IDEAL^{(i)}(k,x;r)\}_{k\in N}$$

Where $S = (S_1, S_2, S_3)$, $S_i = Sim_i(A_i)$, and r is uniformly random.

4 Protocol

In this section, we formally define our black box outsourcing protocol. For a graphical representation, see Figure 1.

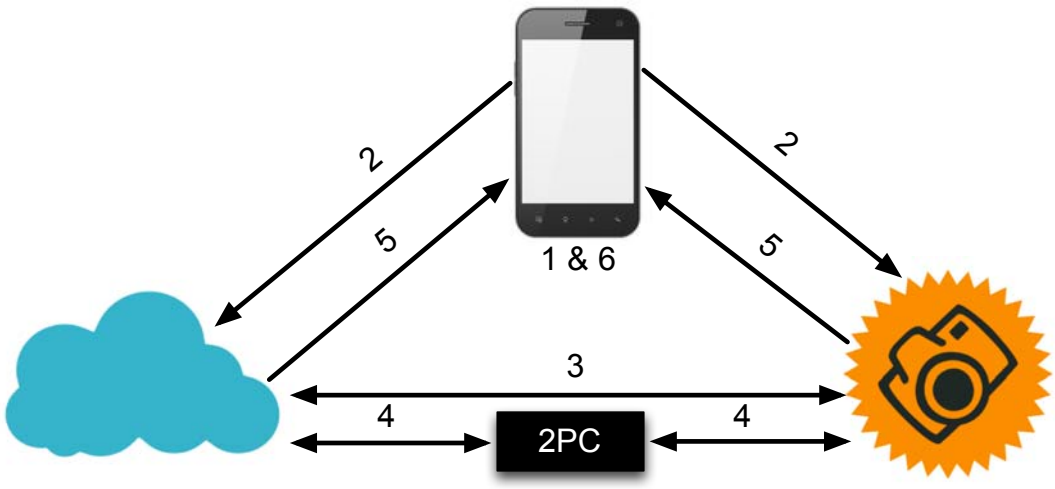


Figure 1: The complete black box outsourcing protocol. Note that the mobile device performs very little work compared to the application server and the Cloud, which execute a two-party SMC (2PC) protocol.

4.1 Participants

- SERVER: the application or web server participating in a secure computation with the mobile device. This party provides input to the function being evaluated.
- MOBILE: the mobile device accessing SERVER to jointly compute some result. This party also provides input to the function being evaluated.
- CLOUD a Cloud computation provider tasked with assisting MOBILE in the expensive operations of the secure computation. This party executes a two-party SMC protocol in a black-box manner with SERVER, but does not provide an input to the function being evaluated.

4.2 Overview

The outsourcing protocol can be informally broken down as follows: first, the mobile device prepares its input by encrypting it and producing a MAC tag for verifying the input is not tampered with before it is entered into the computation. Since the application server and Cloud are assumed not to collude, one party receives the encrypted input, and the other party receives the decryption key. Both of these values are input into the secure two-party computation, and are verified within the secure two-party protocol using the associated MAC tags (see Figure 2). If the check fails, the protocol outputs a failure message. Otherwise, the second phase of the protocol, the actual evaluation of the SMC program, takes place. The third and final phase encrypts and outputs the mobile device's result to both parties, who in turn deliver these results back to the mobile device. Intuitively, since our security model assumes that the application server and the Cloud are never simultaneously malicious, at least one of these two will return the correct result to the mobile device. From this, the mobile will detect any tampering from the malicious party by a discrepancy in these returned values, eliminating the need for an output MAC. If no tampering is detected, the mobile device then decrypts the output of computation.

4.3 Protocol

Common Input: All parties agree on a computational security parameter k, a message authentication code (MAC) scheme $(Gen(), Mac(), Ver())$, and a malicious secure two-party computation protocol $2PC()$. All parties agree on a two-output function $f(x, y) \; ! \; f_m, f_s$ that is to be evaluated.

Private Input: MOBILE inputs x while SERVER inputs y. We denote the bit length of a value as $|x|$ and concatenation as $x||y$.

Output: SERVER receives f_s and MOBILE receives f_m.

1. Input preparation: MOBILE generates a one-time pad k_{fm} where $|k_{fm}| = |f_m|$. Mobile then generates two MAC keys $v_s = Gen(k)$ and $v_c = Gen(k)$. Fi-

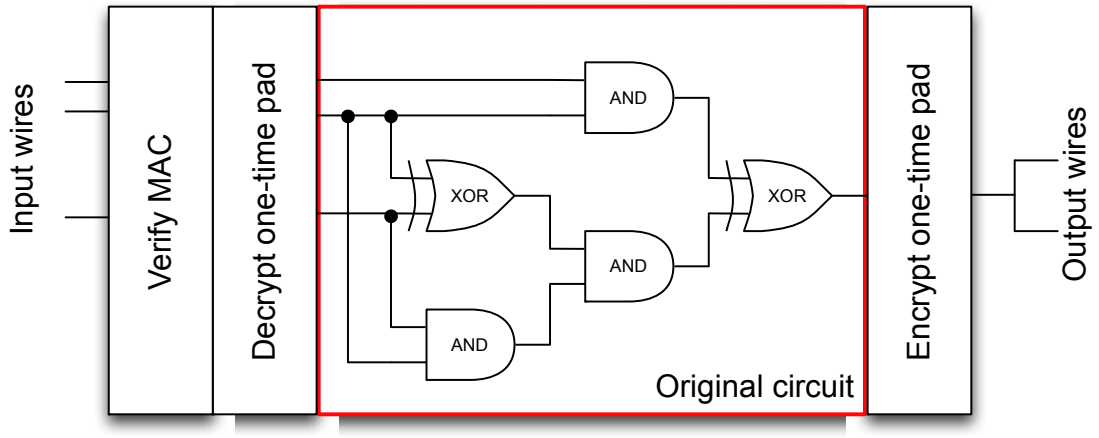


Figure 2: The process of augmenting a circuit for outsourcing. The original circuit is boxed in red. Essentially, we require that the mobile device's input be verified using a MAC and decrypted using a one-time pad before it is input into the function. After the result is computed, it must be re-encrypted using a one-time pad and delivered to *both* parties to guarantee that the mobile device will detect if either party tampers with the result.

nally, MOBILE generates a one-time pad k_m where $|k_m| = |x| + |k_{fm}|$.

2. Input delivery: MOBILE encrypts its input as $a = (x||k_{fm}) \oplus k_m$. It then generates two tags $t_s = Mac(a||v_c, v_s)$ and $t_c = Mac(k_m||v_s, v_c)$. MOBILE delivers a, v_c, and t_s to SERVER and k_m, v_s, and t_c to CLOUD.

3. Augmenting the target function (Algorithm 1): All parties agree on the following augmented function $g(y, a, v_c, t_s; k_m, v_s, t_c)$ to be run as a two-party SMC computation:

   (a) If $Ver(a||v_c, t_s, v_s) \neq 1$ or $Ver(k_m||v_s, t_c, v_c) \neq 1$ output $?$.

   (b) Set $x||k_{fm} = a \oplus k_m$

   (c) Run the desired function $f_s, f_m = f(x, y)$

   (d) Set output values $o_s = f_s$ and $o_m = f_m \oplus k_{fm}$

   (e) Output $o_s||o_m$ to SERVER and o_m to CLOUD

4. Two-party computation: SERVER and CLOUD execute a secure two-party computation protocol $2PC(g(); y, a, v_c, t_s; k_m, v_s, t_c)$ evaluating the augmented function.

5. Output verification: CLOUD delivers its output from the two-party computation, o_m to MOBILE. SERVER also delivers the second half of its output o_m^0 to MOBILE. MOBILE verifies that $o_m = o_m^0$.

6. Output recovery: SERVER receives output $f_s = o_s$ and MOBILE receives output $f_m = o_m \oplus k_{fm}$

```
Input  : CLOUD inputs k_m, v_s, t_c and SERVER inputs
         y, a, v_c, t_s
Output: CLOUD receives o_m and SERVER receives
         o_s||o_m
if Ver(a||v_c, t_s, v_s) ≠ 1 then
    return ?
else if Ver(k_m||v_s, t_c, v_c) ≠ 1 then
    return ?
else
    x, k_fm = a ⊕ k_m
    f_m, f_s = f(x, y)
    o_s = f_s(x, y)
    o_m = f_m(x, y) ⊕ k_fm
end
```

Algorithm 1: The augmented function

5 Security

Our black box outsourcing protocol is secure under the following theorem satisfying the security definition from Section 3:

Theorem 1. *The black box outsourced two-party protocol securely computes a function $f(x, y)$ in the following two corruption scenarios: (1) Any one party is malicious and non-cooperative with respect to the rest of the parties; (2) The Cloud and the mobile device are malicious and colluding, while the application server is semi-honest.*

Note that these scenarios correspond exactly with the

corruption scenarios in [7], and that the previous protocols described in [24] and [8] are only secure in corruption scenario (1). We outline sketches of the security proof here, with a complete proof in Appendix A.

5.1 Malicious Cloud or Application Server

The main idea behind the security in these two settings is that for whichever party is corrupted, we can rely on the other party to behave semi-honestly. Based on the security of the underlying two-party protocol, this ensures both that the augmented functionality is correctly evaluated and that the mobile device will receive unmodified output from one of the parties. Thus, the MAC on the input and the comparison of the output values prevents either party from modifying the Mobile device's private values. Furthermore, unlike the dual execution model by Huang et al. [19] where the output comparison leaks one bit of input, our output comparison is composed of two copies of the mobile output produced from a single, malicious-secure execution of the augmented circuit. Because of this, any discrepancy in the comparison only reveals that either the Cloud or Application Server tampered with the output prior to delivering it to the mobile device.

In the ideal world, the simulator works roughly as follows: begin the black box protocol with random inputs. Then, invoke the simulator for the underlying two-party scheme S_{2PC} to recover the input of the malicious party and delivers that input to the trusted third party. Finally, S_{2PC} simulates the output $f(x,y)$. After running all consistency verifications, the simulator either sends an early termination signal to the trusted third party or completes the protocol normally.

5.2 Malicious Mobile Device

Because the mobile device simply provides MAC tagged input and receives its output after executing the two-party protocol, there is very little it can do to corrupt the computation besides providing invalid inputs that would simply cause the computation to terminate early. The simulator in this scenario accepts the mobile device's prepared inputs. Given both the Cloud and the Application Server's halves of the mobile device's input, the simulator can recover the necessary input by decrypting the one-time pad. If either of the MAC tags does not verify or if the mobile device terminates early, the simulator also terminates. Otherwise, it invokes the trusted third party to receive $f(x,y)$ and returns the result to the mobile device.

5.3 Malicious Mobile Device and Cloud

In this scenario, the security of our black box protocol simply reduces to the security of the underlying two-party scheme. The simulator in the ideal world accepts the input from the Mobile Device, then invokes the simulator of the underlying two-party SMC scheme S_{2PC} to recover the values input by the Cloud. Using these values combined with the values provided by the Mobile Device, the simulator can recover the Mobile input. If any of the verification checks within the augmented functionality fail, the simulator terminates. Otherwise, it delivers the recovered input to the trusted third party, and finishes S_{2PC} delivering the output of computation correctly formatted using the one-time pads recovered from the Cloud's input by S_{2PC}.

6 Performance Evaluation

To demonstrate the practical efficiency of our black box outsourcing protocol, we implemented the protocol and examined the actual overhead incurred by the overhead operations. We initially considered comparing our black box protocol to existing implementations of outsourcing protocols [24, 8, 7]. However, these existing protocols are built on fixed underlying SMC techniques. As new protocols for two-party SMC are developed, the plug-and-play nature of our protocol allows for these new techniques to be applied, which would provide a different comparison for each underlying protocol. Instead, we chose to compare the overhead execution costs of our black box protocol to performing the same computation in the underlying two-party protocol. Because the mobile device computation requires seconds or less to execute, we focus our attention on the cost at the two executing servers. This performance analysis demonstrates two key benefits of our protocol. First, it gives a rough overhead cost for an entire class of two-party SMC protocols (in our case, garbled circuit protocols). Second, it allows us to demonstrate that our outsourcing technique allows a mobile device with restricted computational capability to participate in a privacy-preserving computation in approximately the same amount of time as the same computation performed between two servers. Essentially, we show that our protocol provides a mobile version of any two-party SMC protocol with nominal overhead cost to the servers. This is a novel evaluation methodology not used to evaluate previous black box SMC constructions, and provides a more intuitive estimate for performance when applying a new underlying SMC construction.

Program Name	SS13 Total	BB Total	Increase	SS13 Non-XOR	BB Non-XOR	Increase
Dijkstra10	259,232	456,326	1.8x	118,357	179,641	1.5x
Dijkstra20	1,653,542	1,949,820	1.2x	757,197	849,445	1.1x
Dijkstra50	22,109,732	22,605,018	1.0x	10,170,407	10,324,317	1.0x
MatrixMult3x3	424,748	1,020,196	2.4x	161,237	345,417	2.1x
MatrixMult5x5	1,968,452	3,360,956	1.7x	746,977	1,176,981	1.6x
MatrixMult8x8	8,069,506	11,354,394	1.4x	3,060,802	4,075,082	1.3x
MatrixMult16x16	64,570,969	77,423,481	1.2x	24,494,338	28,458,635	1.2x
RSA128	116,083,727	116,463,648	1.0x	41,082,205	41,208,553	1.0x

Table 1: A comparison of the original function size to the augmented outsourcing circuit. As the size of the original circuit grows, the increase in gates incurred by our outsourcing technique becomes vanishingly small.

6.1 System Design

Our implementation of the black box outsourcing protocol uses the two-party garbled circuit protocol developed by shelat and Shen [41] as the underlying two-party SMC protocol. We selected this protocol because it is among the most recently developed garbled circuit protocols and it has the most stable public release. We emphasize that it is possible to implement our outsourcing on *any* two-party SMC protocol, such as the recent protocols developed to reduce the cost of cut-&-choose [20, 29]. We implement our MAC within the augmented circuit using AES in cipher-block chaining mode (CBC-MAC), as the AES circuit is well-studied in the context of garbled circuit execution. This MAC implementation adds an invocation of AES per 128-bit block of input. Using the compiler developed by Kreuter et al. [28], the overhead non-XOR gate count in the augmented circuit based on input size is $(\frac{|x|15686}{128})$ for input x. We provide exact gate counts with overhead measurements for each tested application in Table 1. Our code will be made available upon publication.

6.1.1 Testbed

Our experiments were run on a single server equipped with 64 cores and 1 TB of RAM. For each execution, the application server and cloud were run as 32 processes communicating using the Message Passing Interface (MPI) framework. The mobile device used was a Samsung Galaxy Nexus with a 1.2 GHz dual-core ARM Cortex-A9 processor and 1 GB of RAM, running Android version 4.0. The mobile device communicated with the test server over an 802.11n wireless connection in an isolated network environment. We ran each experiment 10 times and averaged the results, providing 95% confidence intervals in all figures.

Symmetric	Asymmetric	Bandwidth (bits)
9	0	$2(\|x\|+2k)+4(\|o_m\|)$

Table 2: The total operations and bandwidth required at the mobile device. Recall that $|x|$ is the length of the mobile input in bits, k is the security parameter, and $|o_m|$ is the length of the mobile output in bits. When measured with the total protocol execution time, these operations are lost in the confidence intervals.

6.1.2 Test applications

We selected a representative set of test applications from previous literature [7, 28, 41, 27] to examine the performance of our protocol over varying circuit and input sizes. We use all applications as implemented by Kreuter et al. [28] except for Dijkstra's algorithm, which was implemented by Carter et al. [8].

1. Dijkstra: this application accepts a weighted graph from one party and two node indices from the other party (i.e., start and end nodes), and calculates the shortest path through the graph from the start to the end node. We consider n-node graphs with 16 bit edge weights, 8 bit node identifiers, and a maximum degree of 4 for each node. We chose this problem as a representative application for the mobile platform.
2. Matrix Multiplication: this application accepts a matrix from both parties and outputs the matrix product. We consider this application for input size n, where each matrix is an $n \times n$ matrix of 32-bit integers. This test application demonstrates protocol behavior for increasing input sizes.

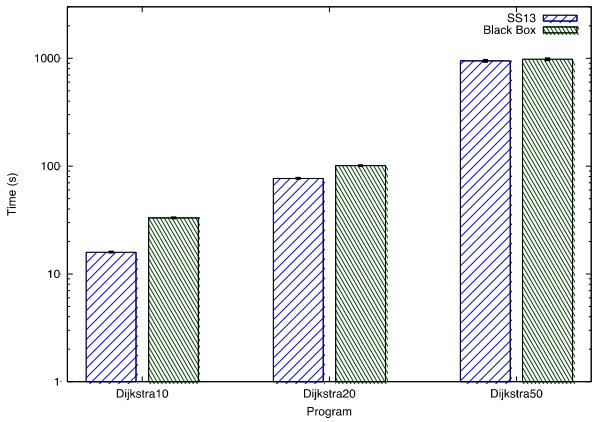


Figure 3: Dijkstra execution time in seconds. Note that for the largest input size, the execution overhead of outsourcing is almost non-existent.

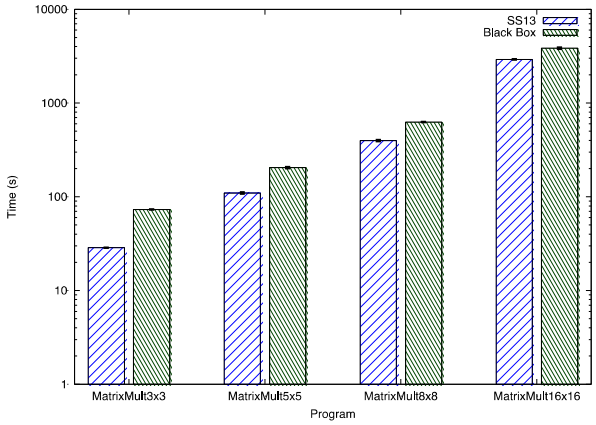


Figure 4: Matrix multiplication execution time in seconds. Note that the execution overhead still diminishes even as the mobile input size increases.

3. RSA: this application accepts a modulus N and an exponent e from one party, and a message x from the other party, and computes the modular exponentiation $x^e \mod N$. We consider input values where each value is 128 bits in length. While this is certainly too short for secure practical use, the size of the circuit provides a good benchmark for evaluating extremely large circuits.

6.2 Execution Time

With the mobile operations reduced to a minimal set, shown in Table 2, our experiments showed a diminishing cost of server overhead as the size of the test application increased. Considering Dijkstra's algorithm in Figure 3 shows that for a graph of 10 nodes, the outsourcing operations incur a 2.1x slowdown from running the protocol between two servers. However, as the number of graph nodes increases to 50, the confidence intervals for outsourced and server-only execution overlap, indicating a virtually non-existent overhead cost. When we compare these results to the gate counts shown in Table 1, we see that as the gate count for the underlying protocol increases, the additive cost of running the input MAC and output duplication amortize over the total execution time. This is to be expected from our predicted overhead of 15686 non-XOR gates for each CBC-MAC block in the input. However, since the mobile input for Dijkstra's algorithm is of a fixed size, we observe that increasing the application server input size does not add to the outsourcing overhead, showing the black box protocol to be more efficient for large circuit sizes with small mobile input.

When we consider a growing mobile input size, we observe the overhead cost of the MAC operation performed on the mobile input. In the matrix multiplication test program, we observed a 2.6x slowdown for the smallest input size of a 3×3 matrix (Figure 4). As in the previous experiment, this overhead diminished to a 1.3x slowdown for the largest input size, but diminished at a slower rate when compared to the circuit size. This is a result of additional AES invocations to handle the increasing mobile input size. However, the reduction in overhead shows that even as input sizes increase, the circuit size is still the main factor in amortizing overhead.

In our final experiment, we considered a massive circuit representing one of the most complex garbled circuit programs evaluated to date. When comparing the outsourced execution to a standard two-party execution, the overhead incurred by the outsourcing operations is almost non-existent, as shown in Table 3. This experiment confirms the trends of diminishing overhead cost observed in the previous two experiments. From this and previous work, we know that evaluating large circuits from mobile devices is *not possible* without outsourcing the bulk of computation. Given that many real-world applications will require on the order of billions of gates to evaluate, this experiment shows that our black box outsourcing technique allows mobile devices to participate in secure two-party computation at roughly the same efficiency as two server-class machines executing the same computation.

6.3 Bandwidth

Because transmitting data from a mobile device is costly in terms of time and power usage, we attempted to minimize the amount of bandwidth required from the mobile device. Thus, the bandwidth used by the mobile device for any given application can be represented as a simple formula, shown in Table 2. Because this band-

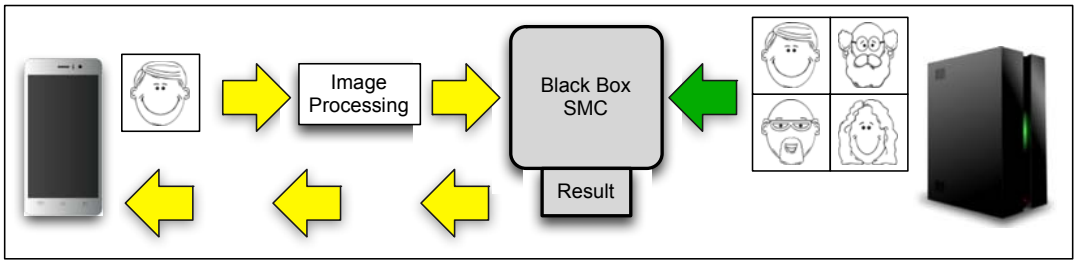


Figure 5: An example of the facial recognition application.

Program Name	SS13	BB	Increase
Dijkstra10	16 ± 1%	33 ± 1%	2.1x
Dijkstra20	77 ± 1%	100 ± 1%	1.3x
Dijkstra50	940 ± 2%	980 ± 2%	1.0x
MatrixMult3x3	28.6 ± 0.8%	73.2 ± 0.5%	2.6x
MatrixMult5x5	110 ± 2%	200 ± 2%	1.9x
MatrixMult8x8	400 ± 2%	627 ± 0.9%	1.6x
MatrixMult16x16	2900 ± 1%	3800 ± 2%	1.3x
RSA128	4700 ± 2%	4900 ± 3%	1.0x

Table 3: Comparing SS13 and Black Box runtime. All times in seconds. Note that as the circuit size increases, the increase in execution time caused by outsourcing becomes insignificant.

Program Name	SS13	BB	Increase
Dijkstra10	2.44×10^9	3.87×10^9	1.6x
Dijkstra20	1.52×10^{10}	1.73×10^{10}	1.1x
Dijkstra50	2.02×10^{11}	2.05×10^{11}	1.0x
MatrixMult3x3	3.43×10^9	7.66×10^9	2.2x
MatrixMult5x5	1.57×10^{10}	2.56×10^{10}	1.6x
MatrixMult8x8	6.43×10^{10}	8.73×10^{10}	1.4x
MatrixMult16x16	5.11×10^{11}	6.01×10^{11}	1.2x
RSA128	8.69×10^{11}	8.72×10^{11}	1.0x

Table 4: Comparing SS13 and Black Box bandwidth usage between the parties performing the generation and evaluation of the garbled circuit. All bandwidth in bytes. Note that the size of the original circuit dominates the bandwidth required between the two servers. As this circuit grows in size, the overhead bandwidth required for outsourcing is amortized.

width is nearly minimal and easily calculated for any test program, we focused our experimentation on examining the bandwidth overhead incurred between the application server and the Cloud.

As in the case of execution time, Table 4 shows an inverse relation between circuit size and overhead cost. Before running the experiment, we predicted that the bandwidth overhead would approximately match the overhead in circuit size shown in Table 1. The experiments confirmed that the actual bandwidth overhead was equal to or slightly larger than the overhead in non-XOR gates in the circuit. The reason for this correlation is twofold. First, the free-XOR technique used in the shelat-Shen protocol allows XOR gates to be represented without sending any data over the network. Thus, adding additional XOR gates does not incur bandwidth cost. Second, in cases where the actual overhead is slightly larger than the circuit size overhead, we determined that the added cost was a result of additional oblivious transfers. These operations require the transmission of large algebraic group elements, so the test circuits which incurred increased overhead from the growth of the mobile input showed a slightly larger bandwidth overhead as well. Ultimately, as in the case of execution time, our experiments demonstrate that the black box outsourcing scheme incurs minimal bandwidth usage at the mobile device with diminishing bandwidth overhead between the application server and the Cloud.

7 Application: Facial Recognition

The growing number of mobile applications available present a wealth of potential for applying privacy-preserving computation techniques to the mobile platform. Carter et al. [8] demonstrated one potential application with their privacy-preserving navigation app, and Mood et al. [35] presented a friend-finding application. We present a third mobile-specific application: facial recognition. In this setting, a secret operative or law enforcement agent carrying a mobile device needs to analyze a photo of a suspected criminal using an international crime database (see Figure 5). The database,

Program Name	Time
FaceRec10	$87.1 \pm 0.9\%$
FaceRec100	$170 \pm 2\%$
FaceRec1000	$1000 \pm 2\%$

Table 5: Runtime results showing the time it takes to determine what the input face is when a database of 10, 100, or 1000 faces is used. Time indicates the total runtime of the garbled circuit part of the computation. All time in seconds. These results demonstrate how outsourcing allows an application designed for desktop-class machines to be efficiently executed from the mobile platform.

managed by an international organization, would compare the photo to their database in a privacy-preserving manner, returning a match if the suspect appears in the database. In this scenario, the agent must keep the query data private to prevent insiders from learning who is being tracked, and the international organization must keep the database private from agents associated with any particular nation.

To implement this application, we use the facial recognition techniques developed for the Scifi protocol of Osadchy et al. [38]. They develop a technique for two servers to perform efficient facial recognition using discrete parameters, which can more easily be manipulated in secure computation protocols. They combine machine learning techniques in a preprocessing phase with a secure online phase that compares the hamming distance of photos represented as bit strings. To demonstrate our application, we implement the online comparison phase of this protocol in our black box outsourcing protocol (the $F_{threshold}$ function in their work). The mobile device provides a 928 bit representation of a photo, while the application server provides a database of representations containing 10, 100, and 1000 faces.

Our results show that given a database of 10 faces, the outsourced protocol can run the online phase in approximately 87 seconds (see Table 5). As the size of the facial database increases, the execution time for comparing across the entire database grows. This growing cost is a result of the large cost of representing the facial database as garbled input. Provided with a two-party SMC protocol that more efficiently computes over large data sets, our black box protocol could be used to move this application from feasible to practical. This demonstrates that an application designed and implemented to run between two servers can be feasibly executed from a mobile device. As new, more heavyweight applications are developed, our technique for outsourcing allows any of those applications to be executed from a mobile device with comparable efficiency to the server platform.

8 Overhead Analysis

In this section, we analyze the overhead incurred from the outsourcing operations and compare this overhead to the construction proposed by Kamara et al. [24]. While the primary focus of the Salus framework is to develop an outsourcing protocol for multiparty computation with a lower complexity than constructing on a two-party SMC protocol, they include a sketch for outsourcing a two-party SMC protocol in a black box manner (but do not implement this protocol). Essentially, their technique is for the mobile device to generate random bit strings that garble the input and output bit values, similar to the technique of garbling inputs and outputs in a Yao garbled circuit. These bit labels, along with the encoded input, are then secret shared between the application server and the Cloud, who execute the computation using a two-party SMC protocol. Although they provide no formal proof, the intuition behind this scheme is that secret sharing provides privacy of the input and output, and the length of the bit labels computationally prevents a malicious player from modifying the mobile device's input. When comparing this technique to our black box outsourcing protocol, we can analyze the overhead incurred in two parts: the input verification and output verification.

8.1 Input comparison

The black box protocol of Kamara et al. requires that the input of the mobile device be expanded by a security parameter k, such that each bit of input is represented by a bit string of length k. This input is then secret shared between both the application server and the Cloud. Within the SMC computation itself, their technique requires the addition of XOR gates to reconstruct the secret shares, and comparisons to ensure that the input labels have not been modified by either party. Since these operations are relatively inexpensive, especially in garbled circuit style protocols, we can say that the major constraint in their protocol is the expansion of the input size. Our protocol, by contrast, expands the input size by two, plus an additive constant, requiring only the addition of a MAC verification key and a MAC tag. For example, the largest tested input in our experiments was 8 KB for the matrix multiplication of 16×16 matrices. Given a security parameter of 80, the Kamara black box technique would expand this input to over 1.2 MB, while our technique only expands the input to approximately 16 KB. However, our addition of MAC operations within the executed circuit requires greater overhead in computation time, dependent upon the MAC scheme used. Since the goal of our

work is to optimize performance for a mobile device, a primary concern is minimizing bandwidth consumed at the mobile device. Specifically, sending and receiving data wireless from a mobile device consumes significantly more power than processor computation, which means that minimizing bandwidth is a priority for maintaining the utility of the device. To meet this resource constraint, we pay a slightly larger overhead in the two-party computation to reduce the bandwidth sent by two orders of magnitude.

8.2 Output comparison

The output verification technique applied by Kamara et al. requires that the output of the two-party SMC protocol be expanded by a security parameter and transmitted to the mobile device in this expanded form. However, they go on to describe how this verification technique can be applied to allow for a fair release of the output to all parties participating in the computation. Our black box technique exchanges this guarantee of fair release for a radically simpler output verification technique, which only requires the output to be expanded by a factor of two and can be verified with a simple comparison at the mobile device. Again, using the example of 16×16 matrix multiplication, our black box technique reduces the output size from approximately 600 KB to 16 KB. This output verification technique is especially beneficial to our mobile setting, where bandwidth consumption is a major consideration for protocol efficiency. In addition, it functions well in situations where the mobile device is the *only* party that receives output from the secure computation, in which case fair release is no longer a necessary concern.

9 Conclusion

The growing popularity of the mobile platform is creating a strong need for privacy-preserving computation in mobile applications. However, as most SMC techniques currently require significant processing and bandwidth resources, secure outsourcing protocols have been developed to assist mobile devices in performing the most expensive cryptographic operations associated with these protocols. In this work, we develop a technique for outsourcing any two-party SMC protocol in a black box manner. Our protocol securely offloads the cost of the SMC protocol to the Cloud, providing maximal efficiency to the mobile device while maintaining strong security guarantees. Our performance evaluation shows that as the complexity of the program being evaluated increases, the cost of outsourcing diminishes. As a result, we enable execution of any SMC protocol from a mobile device at approximately the same efficiency as running the protocol between two servers.

References

[1] ASHAROV, G., LINDELL, Y., SCHNEIDER, T., AND ZOHNER, M. More efficient oblivious transfer and extensions for faster secure computation. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security* (2013).

[2] ATALLAH, M. J., AND FRIKKEN, K. B. Securely outsourcing linear algebra computations. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2010).

[3] BEAVER, D. Server-assisted cryptography. In *Proceedings of the workshop on New security paradigms (NSPW)* (1998).

[4] BELLARE, M., HOANG, V. T., AND ROGAWAY, P. Foundations of garbled circuits. In *Proceedings of the ACM Conference on Computer and Communications Security* (2012).

[5] BLANTON, M., STEELE, A., AND ALISAGARI, M. Data-oblivious graph algorithms for secure computation and outsourcing. In *Proceedings of the ACM SIGSAC Symposium on Information, Computer and Communications Security* (2013).

[6] CARTER, H., AMRUTKAR, C., DACOSTA, I., AND TRAYNOR, P. For your phone only: custom protocols for efficient secure function evaluation on mobile devices. *Journal of Security and Communication Networks (SCN) 7*, 7 (2014), 1165–1176.

[7] CARTER, H., LEVER, C., AND TRAYNOR, P. Whitewash: Outsourcing Garbled Circuit Generation for Mobile Devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)* (2014).

[8] CARTER, H., MOOD, B., TRAYNOR, P., AND BUTLER, K. Secure Outsourced Garbled Circuit Evaluation for Mobile Devices. In *Proceedings of the USENIX Security Symposium* (2013).

[9] DAMGÅRD, I., KELLER, M., LARRAIA, E., PASTRO, V., SCHOLL, P., AND SMART, N. P. Practical covertly secure mpc for dishonest majority or: Breaking the spdz limits. In *Computer Security–ESORICS* (2013).

[10] DAMGÅRD, I., PASTRO, V., SMART, N., AND ZAKARIAS, S. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO* (2012).

[11] DEMMLER, D., SCHNEIDER, T., AND ZOHNER, M. Ad-hoc secure two-party computation on mobile devices using hardware tokens. In *Proceedings of the USENIX Security Symposium* (2014).

[12] FREDERIKSEN, T. K., JAKOBSEN, T. P., NIELSEN, J. B., NORDHOLT, P. S., AND ORLANDI, C. Minilego: Efficient secure two-party computation from general assumptions. In *Advances in Cryptology–EUROCRYPT* (2013).

[13] GENTRY, C. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[14] GOLDREICH, O. *Foundations of Cryptography: Volume 2 Basic Applications*. Cambridge Univ. Press, 2004.

[15] GOLDREICH, O., MICALI, S., AND WIGDERSON, A. How to play any mental game. In *Proceedings of the Annual ACM Symposium on Theory of Computing* (1987).

[16] GORDON, S. D., KATZ, J., KOLESNIKOV, V., LABS, A.-L. B., KRELL, F., AND RAYKOVA, M. Secure Two-Party Computation in Sublinear (Amortized) Time. In *Proceedings of the ACM conference on Computer and communications security (CCS)* (2012).

[17] HUANG, Y., CHAPMAN, P., AND EVANS, D. Privacy-preserving applications on smartphones. In *Proceedings of the USENIX Workshop on Hot Topics in Security* (2011).

[18] HUANG, Y., EVANS, D., KATZ, J., AND MALKA, L. Faster Secure Two-Party Computation Using Garbled Circuits. In *Proceedings of the USENIX Security Symposium* (2011).

[19] HUANG, Y., KATZ, J., AND EVANS, D. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy* (2012).

[20] HUANG, Y., KATZ, J., AND EVANS, D. Efficient secure two-party computation using symmetric cut-and-choose. In *Advances in Cryptology–CRYPTO* (2013).

[21] HUANG, Y., KATZ, J., AND KOLESNIKOV, V. Amortizing garbled circuits. In *Advances in Cryptology–CRYPTO* (2014).

[22] JAKOBSEN, T. P., NIELSEN, J. B., AND ORLANDI, C. A framework for outsourcing of secure computation. In *Proceedings of the ACM Workshop on Cloud Computing Security (CCSW)* (2014).

[23] KAMARA, S., MOHASSEL, P., AND RAYKOVA, M. Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272, 2011. `http://eprint.iacr.org/`.

[24] KAMARA, S., MOHASSEL, P., AND RIVA, B. Salus: A system for server-aided secure function evaluation. In *Proceedings of the ACM conference on Computer and communications security (CCS)* (2012).

[25] KELION, L. Apple toughens icloud security after celebrity breach. `http://www.bbc.com/news/technology-29237469`, 2014.

[26] KERSCHBAUM, F. Collusion-resistant outsourcing of private set intersection. In *Proceedings of the ACM Symposium on Applied Computing* (2012).

[27] KREUTER, B., SHELAT, A., MOOD, B., AND BUTLER, K. PCF: A portable circuit format for scalable two-party secure computation. In *Proceedings of the USENIX Security Symposium* (2013).

[28] KREUTER, B., SHELAT, A., AND SHEN, C. Billion-Gate Secure Computation with Malicious Adversaries. In *Proceedings of the USENIX Security Symposium* (2012).

[29] LINDELL, Y. Fast cut-and-choose based protocols for malicious and covert adversaries. In *Advances in Cryptology–CRYPTO* (2013).

[30] LINDELL, Y., AND PINKAS, B. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the annual international conference on Advances in Cryptology* (2007).

[31] LINDELL, Y., AND PINKAS, B. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the conference on Theory of cryptography* (2011).

[32] LINDELL, Y., AND RIVA, B. Cut-and-Choose Yao-Based Secure Computation in the Online/Offline and Batch Settings. In *Advances in Cryptology?CRYPTO* (2014).

[33] MALKHI, D., NISAN, N., PINKAS, B., AND SELLA, Y. Fairplay–a secure two-party computation system. In *Proceedings of the USENIX Security Symposium* (2004).

[34] MOHASSEL, P., AND RIVA, B. Garbled circuits checking garbled circuits: More efficient and secure two-party computation. In *Advances in Cryptology–CRYPTO* (2013).

[35] MOOD, B., GUPTA, D., BUTLER, K., AND FEIGENBAUM, J. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the ACM conference on Computer and communications security (CCS)* (2014).

[36] MOOD, B., LETAW, L., AND BUTLER, K. Memory-efficient garbled circuit generation for mobile devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)* (2012).

[37] NIELSEN, J. B., NORDHOLT, P. S., ORLANDI, C., AND BURRA, S. S. A new approach to practical active-secure two-party computation. In *Advances in Cryptology –CRYPTO* (2012).

[38] OSADCHY, M., PINKAS, B., JARROUS, A., AND MOSKOVICH, B. Scifi-a system for secure face identification. In *Proceedings of the IEEE Symposium on Security & Privacy* (2010).

[39] PINKAS, B., SCHNEIDER, T., SMART, N. P., AND WILLIAMS, S. Secure two-party computation is practical. In *Advances in Cryptology–ASIACRYPT* (2009).

[40] SHELAT, A., AND SHEN, C.-H. Two-output secure computation with malicious adversaries. In *Proceedings of the Annual international conference on Theory and applications of cryptographic techniques* (2011).

[41] SHELAT, A., AND SHEN, C.-H. Fast two-party secure computation with minimal assumptions. In *Proceedings of the ACM conference on Computer and communications security (CCS)* (2013).

[42] YAO, A. C. Protocols for secure computations. In *Proceedings of the Annual Symposium on Foundations of Computer Science* (1982).

A Proof of Security

Here we provide the formal simulation proof of security for Theorem 1.

A.1 Malicious MOBILE M^*

In the scenario where M^* can adopt an arbitrary malicious strategy, we construct a simulator S_M that, operating in the ideal world, can simulate M^*s view of a real-world protocol execution and can recover M^*s input for delivery to the trusted third party. We construct this simulator and prove it secure with the following hybrid of experiments.

$Hyb1^{(M)}(k,x;r)$: This experiment is identical to $REAL^{(M)}(k,x;r)$ except that the experiment uses the combination of M^*s encrypted input a and k_m to recover the real input $x*$. It verifies the MAC tags t_s and t_c and aborts if either check fails.

Lemma 1. $REAL^{(M)}(k,x;r) \stackrel{c}{\approx} Hyb1^{(M)}(k,x;r)$

Proof. Since the experiment is controlling both CLOUD and SERVER, it can simply decrypt the input x^* using the key k_m. In addition, since the experiment holds both the verification keys, the protocol will terminate in both experiments if the MAC tags are incorrectly constructed. □

$Hyb2^{(M)}(k,x;r)$: This experiment is identical to $Hyb1^{(M)}(k,x;r)$ except that the experiment passes x^* to the trusted third party, and returns the result $f(x^*,y) \oplus k^*_{fm}$ to M^*, where k^*_{fm} is recovered in the previous hybrid.

Lemma 2. $Hyb1^{(M)}(k,x;r) \stackrel{c}{\approx} Hyb2^{(M)}(k,x;r)$

Proof. Because both experiments use the input x^* for computing the result, the output of the function in both worlds is indistinguishable. Furthermore, the recovered output key allows the experiment to present the result to M^* exactly as it would be in a real world execution. □

Lemma 3. *$Hyb2^{(M)}(k,x;r)$ runs in polynomial time.*

Proof. This lemma follows trivially since a real world execution of the protocol runs in polynomial time and each intermediate hybrid adds only constant time operations. □

We conclude the proof by letting S_M execute $Hyb2^{(M)}(k,x;r)$. S_M runs M^* and controls CLOUD and SERVER. S_M terminates the ideal world execution if any consistency checks fail or if M^* terminates at any point, and outputs whatever M^* outputs at the end of the simulation. From Lemma 1-3, S_M proves Theorem 1 when MOBILE is malicious.

A.2 Malicious SERVER S^*

In the scenario where S^* can adopt an arbitrary malicious strategy, we construct a simulator S_S that, operating in the ideal world, can simulate S^*s view of a real-world protocol execution and can recover S^*s input for delivery to the trusted third party. We construct this simulator and prove it secure with the following hybrid of experiments.

$Hyb1^{(S)}(k,x;r)$: This experiment is identical to $REAL^{(S)}(k,x;r)$ except that the experiment prepares the MOBILE input according to the two-party protocol simulator S_{2PC} instead of using the real MOBILE input. It then prepares the new input according to the protocol and delivers the encrypted input and MAC tags to S^*.

Lemma 4. $REAL^{(S)}(k,x;r) \stackrel{c}{\approx} Hyb1^{(S)}(k,x;r)$

Proof. Since the input is blinded by a one-time pad in both experiments, they are statistically indistinguishable. □

$Hyb2^{(S)}(k,x;r)$: This experiment is identical to $Hyb1^{(S)}(k,x;r)$ except that the experiment invokes the simulator of the two-party SMC protocol S_{2PC} instead of running the actual protocol. S_{2PC} is used to recover S^*s actual input y^*. After recovering the full input, If S^* tampers with MOBILE'S input, S_{2PC} simulates $\perp$ and the experiment terminates. Otherwise, the experiment delivers y^* to the trusted third party and simulates the output $f(x,y^*)$ concatenated with a random string o_{rm}.

Lemma 5. $Hyb1^{(S)}(k,x;r) \stackrel{c}{\approx} Hyb2^{(S)}(k,x;r)$

Proof. Based on the security definition of the underlying two-party SMC protocol, we know that a simulator exists that can simulate the protocol in a computationally indistinguishable way, as well as recover the input used by S^*. Based on the correctness guarantee of the two-party SMC protocol in conjunction with the unforgettability guarantee of the MAC protocol, it is computationally infeasible for S^* to modify MOBILE'S portion of the input. Finally, in both experiments the MOBILE output of the computation is blinded by a one-time pad, making the random output statistically indistinguishable from the real output. □

$Hyb3^{(S)}(k,x;r)$: This experiment is identical to $Hyb2^{(S)}(k,x;r)$ except that the experiment prevents the trusted third party from delivering input to the other party if S^* modifies the MOBILE output o_{rm} before returning it.

Lemma 6. $Hyb2^{(S)}(k,x;r) \stackrel{c}{\approx} Hyb3^{(S)}(k,x;r)$

Proof. Based on the correctness guarantee of the two-party SMC scheme and the fact that CLOUD is semi-honest in this scenario, then S^* will be caught in either experiment, and early termination will be the result. □

Lemma 7. *$Hyb3^{(S)}(k,x;r)$ runs in polynomial time.*

Proof. This lemma follows trivially since a real world execution of the protocol runs in polynomial time, the simulator S_{2PC} runs in polynomial time, and all other intermediate hybrid adds only constant time operations. □

We conclude the proof by letting S_S execute $Hyb3^{(S)}(k,x;r)$. S_S runs S^* and controls CLOUD and MOBILE. S_S terminates the ideal world execution if any consistency checks fail or if S^* terminates at any point, and outputs whatever S^* outputs at the end of the simulation. From Lemma 4-7, S_S proves Theorem 1 when SERVER is malicious.

A.3 Malicious CLOUD C^*

In the scenario where C^* can adopt an arbitrary malicious strategy, we construct a simulator S_C that, operating in the ideal world, can simulate C^*s view of a real-world protocol execution and can recover C^*s auxiliary input for delivery to the trusted third party. We construct this simulator and prove it secure with the following hybrid of experiments.

$Hyb1^{(C)}(k,x;r)$: This experiment is identical to $REAL^{(C)}(k,x;r)$ except that the experiment invokes the two-party SMC simulator S_{2PC}, providing random inputs for SERVER and recovering C^*s real input. Finally, simulate a random result o_r at the end of the two-party computation.

Lemma 8. $REAL^{(C)}(k,x;r) \stackrel{c}{\approx} Hyb1^{(C)}(k,x;r)$

Proof. Based on the security definition of the underlying two-party SMC protocol, we know that the simulator S_{2PC} can indistinguishably simulate the two-party execution and recover MOBILE'S MAC tagged one-time pad as input by C^*. Because in both experiments the output of the circuit is blinded by a one-time pad, the outputs in both cases are statistically indistinguishable. □

$Hyb2^{(C)}(k,x;r)$: This experiment is identical to $Hyb1^{(C)}(k,x;r)$ except that if the experiment finds from the recovered input that C^* modified the random key k_m, the experiment terminates.

Lemma 9. $Hyb1^{(C)}(k,x;r) \stackrel{c}{\approx} Hyb2^{(C)}(k,x;r)$

Proof. Based on the correctness guarantee of the two-party SMC scheme and the unforgettability of the MAC scheme, any change to k_m will cause the circuit to output $\perp$, and will cause MOBILE to terminate except for a negligible probability. Thus, termination in both experiments in computationally indistinguishable. □

$Hyb3^{(C)}(k,x;r)$: This experiment is identical to $Hyb2^{(C)}(k,x;r)$ except that if the experiment aborts if C^* modifies the output string o_r.

Lemma 10. $Hyb2^{(C)}(k,x;r) \stackrel{c}{\approx} Hyb3^{(C)}(k,x;r)$

Proof. Because SERVER is semi-honest and will not tamper with MOBILE'S output, in both hybrids C^* will be caught for tampering with the output and result in an abort of the protocol. □

Lemma 11. $Hyb3^{(C)}(k,x;r)$ *runs in polynomial time.*

Proof. This lemma follows trivially since a real world execution of the protocol runs in polynomial time, the simulator S_{2PC} runs in polynomial time, and all other intermediate hybrid adds only constant time operations. □

We conclude the proof by letting S_C execute $Hyb3^{(C)}(k,x;r)$. S_C runs C^* and controls SERVER and MOBILE. S_C terminates the ideal world execution if any consistency checks fail or if C^* terminates at any point, and outputs whatever C^* outputs at the end of the simulation. From Lemma 8-11, S_C proves Theorem 1 when CLOUD is malicious.

A.4 Malicious MOBILE and CLOUD MC^*

In the final scenario, the colluding parties MC^* can adopt an arbitrary malicious strategy against SERVER. The simulator S_{MC} that proves security in this scenario is essentially the two-party SMC simulator S_{2PC} with one small change. Rather than completely recovering MC^*s input from the simulator, the experiment must combine the malicious MOBILE input $a^*||v_s^*||t_s^*$ with the input recovered by S_{2PC} to learn the real input x^* that is to be delivered to the trusted third party. Once this real input is retrieved, it simulates the result $f(x^*,y)$ exactly as S_{2PC} does. Since the added operations are constant time and S_{2PC} runs in polynomial time, we have that S_{MC} proves Theorem 1 when both MOBILE and CLOUD are malicious and colluding. Note that, as in the underlying two-party SMC scheme, this scenario does *not* guarantee that the output will be released fairly to SERVER. However, it does guarantee privacy and correctness of the output.

Frigate: A Validated, Extensible, and Efficient Compiler and Interpreter for Secure Computation

Abstract

Recent developments in secure computation have led to significant improvements in efficiency and functionality. These efforts created compilers that form the backbone of practical secure computation research. Unfortunately, many of these artifacts are incorrect and unstable, leading to demonstrably erroneous results. We address these problems and present Frigate, a principled compiler and fast circuit interpreter for secure computation. To ensure correctness we apply best practices for compiler design and development, including the use of standard data structures, helpful negative results, and structured validation testing. Our systematic validation tests include checks on the internal compiler state, combinations of operators, and the examination of edge cases based on widely used techniques and errors we have observed in other work. This produces a compiler that builds correct circuits, is efficient and extensible. Frigate creates circuits with gate counts comparable to previous work but does so with compile time speedups as high as 894x compared with the best results from previous work. By creating a validated tool, our compiler will allow future secure computation implementations to be developed quickly and correctly.

1 Introduction

Secure Multiparty Computation (SMC) has long been regarded as a theoretical curiosity. First proposed by Yao [44], SMC allows two or more parties to compute the result of a function without exposing their inputs. The identification of such primitives was groundbreaking, creating opportunities by which untrusting participants could calculate results of mutual interest without requiring all individuals to identify a mutually trusted third party. Unfortunately, it would take more than 20 years before the creation of the first SMC compiler, Fairplay [30], demonstrated that these heavyweight techniques were remotely practical.

The creation of the Fairplay compiler ignited the research community. In the following decade, SMC compilers improved performance by multiple orders of magnitude, significantly reduced bandwidth overhead, and allowed for the generation and execution of circuits composed of tens of billions of gates [32, 25, 16, 24]. While these efforts have incorporated a number of novel elements to achieve the above advances, they all fail in two critical areas. Specifically, as we will demonstrate, these compilers are often unstable and, when they do manage to work, regularly produce outputs that generate incorrect results. Accordingly, the integrity of the results computed by each of these systems is questionable, making their usefulness in practical SMC low.

In this paper, we present Frigate, an SMC compiler developed using design and testing methods from the compiler community. We name our compiler after the naval vessel, known for its speed and adaptability for varying missions. Our compiler is designed to be validated through an extensive battery of testing all facets of its operation, modular and extensible to support a variety of research applications, and faster than the state of the art circuit compilers in the community. In addition, the frigate's use as an escort ship parallels the potential for our compiler to facilitate continued secure computation research. Our contributions are as follows:

- **Demonstrate systemic problems in the most popular SMC compilers:** We apply differential testing on the five popular and available SMC compilers, and demonstrate a range of stability and output correctness problems in them all.

- **Design and Implement Frigate:** Our primary goal in creating Frigate is correctness, which we attempt to achieve through the use of principled and simple design, careful type checking and comprehensive validation testing. We use lessons learned from our study to develop principles for others to follow.

- **Dramatically improve compiler and interpreter performance:** The result of our efforts is not simply correctness; rather, because of our simple design, we demonstrate markedly reduced compilation (by as much as 894x) and interpretation (by over 850x) when compared to currently available systems. As such, our results demonstrate that principled design create correct SMC systems while still allowing high performance.

The remainder of the paper is organized as follows: Section 2 provides readers with a background in SMC; Section 3 introduces techniques used to validate correctness; Section 4 describes the results of our correctness analysis of existing compilers; Section 5 defines our principles for compiler design; Section 6 presents the Frigate compiler and circuit interpreter; Section 7 presents the results of our performance tests comparing Frigate to the most widely used SMC compilers; Section 8 discusses related work; Section 9 provides our final thoughts.

2 Background

Since SMC was originally conceived, a variety of different techniques have been developed. Recent work has demonstrated that each technique can outperform the others in different setups (e.g., number of participants, available network connection, type of function being evaluated) [17, 6, 38]. In this work, we focus specifically on the garbled circuit construction developed by Yao [44]. This protocol has been shown to perform optimally for two-party computation of functions that can be efficiently represented as Boolean circuits. While our experimental analysis examines the performance of the compiler in the context of garbled circuits, it is critical to note that this compiler can be used with *any* SMC technique that represents functions as Boolean circuits.

2.1 Garbled circuits

The garbled circuit construction provides an interactive protocol for obliviously evaluating a function represented as a Boolean circuit. It involves at least two parties: the first party, the *generator*, is responsible for garbling the circuit to be evaluated such that the input, output, and intermediate wire values are obscured. The second party, the *evaluator*, is responsible for obliviously evaluating the garbled circuit with garbled input values provided by both parties.

For each wire i in the garbled circuit, the generator selects random encryption keys k_i^0, k_i^1 to represent the bit values "0" and "1" for each wire in the circuit. Given these garbled wire labels, each gate in the circuit is represented as a truth table (while each gate may have an arbitrary number of input wires, we assume each gate has two inputs without loss of generality). For a gate executing the functionality $\star$ with input wires i and j and output wire k, the generator encrypts each entry in the truth table as $Enc((k_{b_i}^i, k_{b_j}^j), k_{b_i \star b_j}^k)$ where b_i and b_j are the logical bit values of wires i and j. After permuting the entries in each truth table, the generator sends the garbled circuit, along with the input wire labels corresponding to his input, to the evaluator. Given this garbled representation, the evaluator can iteratively decrypt the output wire label for each gate. Once the evaluator possesses wire labels for each output wire, the generator can reveal the actual bit value mapped to the output wire labels received.

To initiate evaluation, the evaluator must hold garbled representations of both parties' input values. However, since the evaluator does not know the mapping between real bit values and garbled wire labels, an oblivious transfer protocol is required to allow the evaluator to garble her own input without revealing it to the generator. Essentially, for each bit in the evaluator's input, both parties execute a protocol that guarantees the evaluator will only learn one wire label for each of her input bits, while the generator will not learn which wire label the evaluator selected.

This protocol guarantees privacy of both parties' inputs and correctness of the output in the semi-honest adversary model, which assumes that both parties will follow the protocol as specified, and will only try to learn additional information through passive observation. When adversaries can perform arbitrary malicious actions, a number of additional checks must be added to ensure that neither party can break the security of the protocol. These checks are designed specifically to prevent tampering with the evaluated function, providing incorrect or inconsistent inputs, or corrupting the values output by the garbled circuit protocol.

2.2 Circuit Compilers

Execution systems for garbled-circuit secure computation require functions that are represented as Boolean circuits. Due to this requirement, there have been several compilers created to generate the circuit representations of common functions used to test this type of computation. These compilers take higher-level languages as input and transform them into a circuit representation. Writing the circuit files directly without using a compiler is tedious, inefficient, and will most likely result in incorrect circuits as they can have billions of gates.

3 Compiler Correctness

One of our main motivations for developing a principled compiler was the varying and unstable state of the ex-

isting research compiler space. Garbled circuit research has made significant advances in the past several years, which is largely due to a set of circuit compilers that have been commonly used to generate test applications for a significant number of protocols. Given our years of experience, we know the reliability of these results is suspect in many cases due to common errors we have found in these compilers. To facilitate continued advances in this research space, a foundational compiler with reliable performance is a critical tool. Without it, researchers will be forced to either use existing compilers, which we show are unreliable, or develop their own compilers, which is time-consuming and slows research progress. To demonstrate the need for a new and correct compiler that is openly available for the community, we examined correctness issues with the most common compilers used in garbled circuit research.

We define the *correctness* of a complier implementation using two criteria: (1) any valid program in the language can be successfully compiled, and (2) the compiler creates the correct output program based on the input file. There are two methods used to demonstrate compiler correctness: formal methods for validation and verification, and validation by testing.

3.1 Formal Verification

The concept of a verifying compiler was identified as a grand challenge by Tony Hoare in 2003 [15] due to the significant complexity in design and implementation. Since that time, the primary example of a formally verified compiler has been CompCert [26]. The development and rigorous proof of each formalized component of the compiler was an immense undertaking. However, despite the amount of time and formal verification that went into CompCert, it was demonstrated that the formal verification used in CompCert was only able to ensure correctness in select components of the compiler. When tested with Csmith [43], there were still errors found that demonstrated the limitations of formal verification. In addition, formal verification of compiler transformations and optimizations is still very much an open research area [29, 33]. Techniques such as *translation validation* [37, 34, 41] focus on the formal validation of a compiler's correctness through the use of static analysis techniques to ensure that two programs have the same semantics, and are designed to attempt to deal with the reality of legacy compilers. They have their limitations as well, particularly within the context of secure multiparty computation compilers that have not adopted any particular standard for intermediate representations. As a result, the semantic model must be adapted for every compiler implementation, and any changes in the compiler require changes to the model.

Based on these limitations and the impracticality of applying formal verification, we instead apply validation techniques that are the standard method for ensuring the correctness of compilers.

3.2 Validation By Testing

Validation by testing demonstrates that a compiler is correct through extensive unit testing. This is by far the most common technique used in practice to ensure compiler correctness. While testing for correctness can miss some errors in compiling specific cases, it provides a practical level of assurance that is sufficient for the vast majority of applications. Validation tests are designed by examining how to test the largest possible number of programs a compiler can generate.

There are many existing validation tests [13, 11, 42] and test suites [4, 1]. The validation tests used by ARM [1] and SuperTest [4] provide a description of the procedures they use to validate the vast majority of possible program cases. However, these suites are language-specific, often developed to find errors in popular tools such as *gcc* and *LLVM*. To date, there have not been existing validation tools designed to examine secure computation compilers. As a result, we developed our own set of validation tests based on the techniques used by these tools. Our tests, like the test suites of ARM and SuperTest, explore the possible statements and effects of those statements.

In our case, hand written tests are preferred over automatically generated tests due to the fact we can examine the compiler source directly. In addition, a different fuzz generator would have to be created for each input language.

Our tests follow the concept of testing the state space of the compiler starting with broad examination of operators and expressions, then refining the tests to consider common special cases. Our tests proceed through five phases.

1. Attempt possible syntactic possibilities and print out the results. This shows that the compiler reads in programs correctly and verifies the internal program state is correct.
2. Beginning from the simplest operation to validate correctness (i.e., outputting a constant) test each operator in the language and each control structure to ensure it outputs the correct result.
   (a) Test the different possible primitive types and declarations.
   (b) Test each operator as to whether it creates the correct output circuit.
   (c) Test each control structure by itself.

(d) Test function calls, parameters, and return statements. Verify that parameters can be used inside of their functions and that return statements work correctly. Also perform tests for where different types are used as input parameters and return values.

3. Validate all the different *paths* for how data can be input into operations. Demonstrate that different control structures work correctly together. Or, as put by SuperTest [4], "Systematically exploring combinations of operators, types, storage classes and constant values."

   (a) Test if the operator deals correctly with the possible types of data that can be input as an operand.

   (b) Test different types of control structures nested within each other.

   (c) Test each operator under *if* conditionals with emphasis on operators that change variable values such as assignment (=), increment (++), and decrement (–).

4. Test edge cases in programs.

   (a) Verify that empty functions that they do not crash on definition or call.

   (b) Test array access and how arrays (and like operators) deal with edge cases, i.e., out of bounds, minimum, and maximum values.

   (c) Ensure known weaknesses in past compilers are tested to determine whether these vulnerabilities appear in other compilers.

5. Perform testing to verify each previously found error was not re-added to the final implementation.

At the conclusion of these tests we have covered the state space in a compiler. We have tested (1) the correctness of each mini-circuit an operator uses, (2) the ways data can come into each operator, (3) the base and nested rule for each construct (*if* statements, *for* loops, arrays declarations), (4) various edge cases.

4 Survey of Existing Compilers

Using the test procedures described in the previous section, we set out to quantify the common problems in existing secure computation compilers. With each compiler, we found failures that would corrupt common test applications for secure computation protocols.

4.1 Comparison Compiler Information

Fairplay: Fairplay [30] was the first compiler to be used for practical research in secure computation. Fairplay's input format is SFDL, a custom hardware description, and the output is SHDL (simply a gate list in ASCII). We selected this compiler since it initiated extensive practical research on improvements.
PAL: We selected the PAL [32] compiler as it was the first compiler designed for low-memory devices using an efficient intermediate representation. It also takes in Fairplay's SFDL and outputs SHDL. It is dramatically more memory efficient than Fairplay and is able to compile much larger programs, but lacks optimizations used in recent garbled circuit protocols.
KSS: We examined the compiler from Kreuter *et al.* [25], hereon referred to as KSS. This compiler takes a hardware specific language as input and outputs a gate list in binary format. We chose KSS since it formed the basis of multiple recently-published works.
CBMC: The recently published CBMC-GC compiler [16] (hereon CBMC) used a bounded model checker to compile a circuit program. This compiler takes a C file as input and outputs a condensed gate list (in ASCII). Because CBMC is the only compiler that can compile programs written in ANSI C, it is commonly used in other garbled circuit research. For this reason, we included it in our comparison.
PCF: The PCF compiler [24], was created in order to have a condensed output format while being efficient. It takes in LCC bytecode as an input language and transforms it into a PCF file (ASCII). This file describes a circuit in a condensed format; a circuit interpreter is used to get each gate in turn. We selected PCF since it has been used to generated some of the largest circuits.[1]

4.2 Analyzing Compiler Correctness

We separate our analysis of previous compilers into two areas: (a) errors in the compiler, and (b) inefficiencies in the system. We only note an error when the original program was valid; if the compiler crashes due to an incorrect program we do not consider it a correctness issue. However, we did find that most of the compilers lacked helpful error messages when an invalid program was provided as input.

4.2.1 Fairplay

Errors: Fairplay cannot output unoptimized constants and sometimes breaks when it encounters single depth *if* statements [32]. In addition, it frequently fails to parse

[1] We use PCF at the time of publication instead of an unpublished newer version (PCF2) as PCF appears to be substantially more stable.

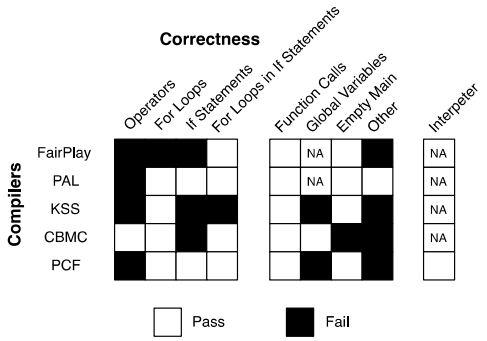


Figure 1: Summary of the correctness results. Fairplay, PAL, and KSS do not have a complex interpreter.

for loops with *if* statements nested within. This is evidence of a failure to properly validate the compiler, as a set combination of operations consistently causes errors.
Inefficiencies and Limitations: Fairplay's compilation process and circuit representation is the most inefficient we tested. The output circuit files are in plaintext, making them significantly larger than necessary. While this facilitates manual inspection of the circuit file, it made the storage requirements for these circuits far too costly for practical use. These inefficiencies and errors imply that Fairplay is only capable of compiling very simple programs that are too small to be relevant in the real world. For example, the AES circuit is a standard benchmark for modern secure computation systems. Fairplay terminates with an out-of-memory error when trying to compile AES even when given over 50GB RAM.

4.2.2 PAL

Errors: PAL encounters problems when *struct*s are used. This appears to be an issue with the compiler's front-end, and is indicative of insufficient validation for that function within the language.
Inefficiencies and Limitations: Like Fairplay, PAL circuits are compiled into plaintext, which results in much larger circuit sizes than using a binary format. PAL also has some problem-size limitations, and fails to compile very large programs. While the templating concept proposed in this work is still useful in building secure computation compilers, this compiler suffers from many of the same inefficiencies as Fairplay, and is not useful for compiling circuits of practical size.

4.2.3 KSS

Errors: The KSS compiler has a number of correctness issues. Nested *if* statements consistently cause errors in the output circuits. Further, *for* loops used within *if* statements also cause the compiler to fail with regularity. In at least one case, we found that the generated circuit can be incorrect because of errors in the optimization phase of compilation (we came up with a work around to this error by XORing in "0".). Finally, we discovered that a variable used inside of a function and then outside (i.e., a global variable defined later in the code) can lead to incorrect behavior as to what the output file will actually do.

Upon examining the architecture of this compiler, we discovered that the steps in compilation do not follow standard good practice in compiler constructions. Without a simple-to-parse AST representation of the program, careful validation of the compiler would be difficult.
Inefficiencies and Limitations: Rather than reduce the output size using templating, KSS outputs the entire circuit. It also uses a very large amount of hardware-specific code, which makes porting it to other environments an extremely difficult task. While this hardware-specific code provides some efficiency gain on specific platforms, it makes the task of extending the code very complex.

4.2.4 CBMC

Errors: The published version of the CBMC compiler crashes if the program has unused inputs. This can occur when an input is not directly used in an expression but instead only used within a conditional branch that is never evaluated. CBMC crashes if even a single bit of input is not used. CBMC also crashes if an input variable is written within a program rather than at the start. Output variables can also cause compiler errors if used more than once, or if read inside the program. The entire framework crashes if we try to compile circuits, which have no gates (just the input and output pins) demonstrate CBMC cannot take programs that should be trivial to perform correctly. CBMC sometimes compiles successfully when arrays are used as input and sometimes fails.
Inefficiencies and Limitations: CBMC outputs the entire circuit in a plaintext format, which while condensed compared to Fairplay and PAL, is still much larger than using a binary circuit file. The output file format also doesn't map output variables to pins, making developing and debugging an interpreter prone to error.

4.2.5 PCF

Errors: PCF allows global variables but does not allow global values to be initialized with a value i.e., assignment must happen later on in the program. Also, when an array location is addressed out of bounds, each attempt we PCF will fill in the result with "0"s for the variable instead of producing an error message for each test we made. This is extremely dangerous behavior, as it can

lead to hidden and hard-to-detect errors. By default, PCF does not update the number of input and output wires based on the input size of the program being compiled, meaning the output is incorrect.

The *translate* script provided with PCF, which is used to convert LCC bytecode to PCF, can fail on valid files (*e.g.*, the edt.16 (edit distance) code provided in the PCF example circuits). In addition, PCF has input buffer overflow problems where inputs above 2^{14} bits overflow the input-buffers for the two parties. This means that the circuit will always fail upon evaluation. These input size bounds are currently hard-coded into the PCF compiler, not defined by the program being compiled, and must be edited manually in cases where larger inputs are needed. These nonstandard approaches to managing program parameters and data lead to a compiler that is confusing to use and difficult to debug.

Inefficiencies and Limitations: While PCF produces very small output circuits, the interpreter required to parse these circuits is extremely inefficient. Our tests demonstrated that the interpreter can require as many as ten operations to read in a single gate. This overhead is magnified by the fact that each gate is read by the interpreter for *every* circuit that is garbled. For malicious secure execution systems where many copies of the same circuit must be garbled, it is far more efficient to parse the gate once, then garbled the same functionality as many times as are required for protocol security. PCF also produces spurious gates, which add to the circuit complexity and should be removed. As with many other compilers studied, PCF uses plaintext output format, using more storage space than necessary.

5 Compiler Development Principles

Given the deplorable state of secure computation compilers in the research community, we set the primary goal of our work to be the development of structured design practices for secure computation compilers, and to demonstrate the effectiveness of these practices with a new compiler implementation. By examining practices used by the compiler community and combining those best practices with the observed failings of previous secure computation compilers, we have assembled a set of four principles to guide the development of our compiler, Frigate. Through this implementation, we demonstrate that these principles should be considered standard practice when developing new compilers for secure computation applications.

1. *Use standard compiler practices:* Use standard methodology from compilers (lexing, parsing, semantic analysis, and code generation). Use data structures that are described throughout compiler literature (e.g. an abstract syntax tree) [5]. Applying these standard, well-studied constructs allows for straightforward modular treatment of the compiler components when extending the functionality. Furthermore, it allows for application of standard compiler debugging practices.

2. *Validate the compiler output:* All production compilers rely on proper program validation to ensure that the compiler functions correctly. A variety of validation test sets have been developed in both the research community and in industry that can be applied to newly-developed compilers [4, 1, 36].

3. *Handle errors well with helpful error messages:* Many sources describing good compiler practices emphasize the need to produce error messages, also known as negative results (e.g,. [5, 4]). While allowing the compiler to crash silently on an incorrect program does not affect its overall correctness, it severely hampers usefulness.

4. *Simplify the design:* A standard software engineering principle is to avoid erroneous code by using simple designs. This allows for more intuitive debugging when errors do occur, as well as facilitating the addition of future functionality.

6 The Frigate Compiler

To demonstrate the practical effectiveness of our compiler design principles, we designed the Frigate compiler and secure computation language. We also created a fast interpreter to read Frigate's output files efficiently. Our work demonstrates three additional contributions to the state of secure computation compiler research: (1) a new and simplified C-style language with specifically designed constructs and operators for producing efficient Boolean circuit representations; (2) a compiler that produces circuits with orders of magnitude less execution time than previous compilers; and (3) a novel circuit output format that provides an efficient balance between compact representation and speed of interpretation.

6.1 Input Language

To better facilitate the development of programs that can be efficiently compiled into Boolean circuits, we developed a custom C-style language to represent secure computation programs. The language allows for efficiently defining arbitrary bit-length variables that translate readily into wire representation, and restricts operations in a manner that allows for full program functionality without excessive complexity. This minimal set of operations adheres to our fourth design principle of maintaining

Operators	Description
+ - □	signed arithmetic operators
/ %	unsigned arithmetic operators
| ^ & □	bitwise operators
=	assignment operator
== !=	equality test operators
> < <= >=	conditional operators
<< >>	shift operators
<<<	rotate left operator
.	struct operator
[]	array operator
{} {:}	wire operators

Table 1: A table showing the operators in Frigate's input language

simplicity to ensure for easier validation. Our language has control structures for functions, *for* loops, and *if/else* statements. We include the ability to define types of arbitrary length and combination as in SFDL, the language used by Fairplay, combined with an operator that selects some bits from a variable used in the KSS compiler input language. For modularity, we have *#include* statements to allow the use of external files and *#define* to replace a term with an expression. The list of operators in our language is in Table 1, with an example of our input language in Appendix B.

Every program begins with a declaration of the number of parties participating in the computation. Since not every participant is required to provide input or receive output, the input and output types for any subset of the participants may then be specified.

To further maintain simplicity, only two primitive types are defined in our programing language. *int_t* types are numbers defined to a specific bit length while *struct_t* types may consist of *int_t* and *struct_t* types. Developers may specify their own types using these two types and the *typedef* command. These two types can be combined to create any complex data type. To formally define the typing of each operator in our language, we give a selection of typing rules in Figure 2. The remainder of these rules are available in Appendix A.

One feature we were compelled to omit from our language was global variables. We removed this feature after we realized the significant overhead they represent within a Boolean circuit program. Allowing global variables requires keeping track of whether each function is called under an *if* statement and adding a MUX gate every time a global variable wire is assigned a value. Our language is capable of expressing equally functional programs by passing in "global" variables and returning any new values for this variables.

$$\text{Add}\ \frac{\Gamma \vdash t_i : Num_{L_i}}{\Gamma \vdash t_1 + t_2 : Num_{L_i}} \qquad \text{Less}\ \frac{\Gamma \vdash t_i : Num_{L_i}}{\Gamma \vdash t_1 < t_2 : Num_1} \qquad \text{Assn}\ \frac{\Gamma \vdash t_i : T}{\Gamma \vdash t_1 = t_2 : T}$$

$$\text{If-Else}\ \frac{\Gamma \vdash t_i : T \quad \sigma : Num_1}{\Gamma \vdash \text{if } (\sigma)\{t_1\} \text{ else } \{t_2\} : T} \qquad \text{Func-Call}\ \frac{\Gamma \vdash t_i : T_i \quad f : F}{\Gamma \vdash f(t_0 \ldots t_{n-1}) : R}$$

Figure 2: Example typing rules for basic operators and control flow statements

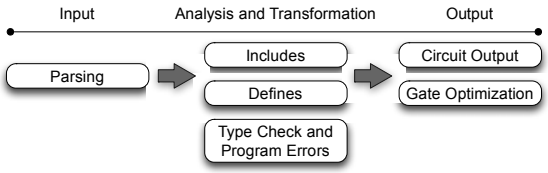


Figure 3: Overall design of the Frigate compiler. There are six separate blocks of the compiler. We have separated blocks into three different stages instead of the traditional two stages.

6.2 Compiler Design

With our input language defined, we next examine the design of the Frigate compiler itself. Written in approximately 20,000 lines of C++, the compiler is designed to be simple enough to validate each output code path and modular for expansion to fit specialized secure computation applications. We plan to make our code available upon publication.

6.2.1 Compilation stages

Frigate represents programs in the standard compiler data structure, the *abstract syntax tree (AST)*. In accordance with our first design principle, this allows for straightforward static analysis and transformation of each program. Each type of operation has its own node where construction, type checking, and output of its subcircuit (among other functions) takes place.

Compilation of a program follows three phases as shown in Figure 3. The input section of Frigate takes in a program and creates an AST representation of the program. We used Flex [2] and Bison [3] to generate the scanner and parser used in this phase. In the second phase, any *#include* statements are replaced with the included file's generated AST. All *#define* statements replace any terms in the AST with a deep copy of the defined expression tree. To conclude this phase, the type checker takes the AST and checks that it is a valid program as defined by Frigate's input language. The final phase of compilation takes in the AST and outputs the circuit while performing gate-level optimizations. If a developer wishes to extend the functionality of Frigate,

this modular phase design allows for additional stages to be inserted in between the existing stages.

6.2.2 Type Checking and Error Output

To satisfy our third design principle, we created our type checker to output detailed error messages to indicate the location and type of error generated by an incorrect program (e.g. *./tests/add.wir, Error line:11 Type "mytype" is used but not defined*). To ensure developers do not include unstable functionality in their programs, Frigate enforces strict type checking that prevents different types from interacting unless those types are different *int_t* types of the same length. A warning is issued in this case.

6.3 Circuit Representation

Previous work in compiler development has demonstrated that it is possible to have either a large yet simple circuit representation that is efficient to parse, or a highly compact circuit representation that incurs a significant cost when it is interpreted by the evaluation program. To strike a balance between these two extremes, we developed a novel circuit representation that is significantly smaller than the simplified circuit representations while still being efficiently parseable. Our output format represents circuits using four elements: a set of input and output calls, gate instructions, function calls, and copy instructions. Our representation of function calls allows us to shrink the output size but still prevents the need for a costly circuit interpreter (more details in Section 6.3.2).

To further improve the efficiency of evaluating Frigate circuits, we designed the compiler to favor XOR gates, as they can be evaluated with fewer operations and do not consume bandwidth when certain garbled circuit protocol optimizations are used [23]. We use the four-XOR, one AND-full adder used by CBMC and PCF.

6.3.1 Output Components

Here we present the detailed components of our circuit representation.

Wires: Each variable is composed of many wires that are allocated as needed with a set address. Each wire exists in either a *used* wire bin or a *free* wire bin. Once a *used* wire is freed it is placed in the *free* bin. Order, as defined by the address of a wire, is not preserved in the *free* wire bin. Our compiler will free the wires it can after each operation.

Wires can exist in one of six states. *ZERO* and *ONE* represent a wire's state as 0 or 1. The *UNKNOWN* state represents wires that depend on input values such that their value cannot be computed at compile time. *UNKNOWN_INVERT* represents an unknown wire but at some point was inverted. *UNKNOWN_OTHER* and *UNKNOWN_INVERT_OTHER* are wires whose values are pointers to another wire value or the inversion of another wire value. By keeping track of inverted states, we can optimize away inverts in some cases.

Gate Output: Given two input wires and a truth table, the *outputGate* function will output a gate and update the state of the output wire. An additional function is called to determine whether the gate is needed or whether it can be short-circuited, i.e., the correct result of the gate can be computed in the compiler. If the gate cannot be short-circuited then the truth table will be adjusted for whether either of the input wires' states are inverted. Finally, the gate will be added to the output.

Function Parameters and Return States: Because we output the gate representation of each function independently a single time, and not reflective of a single function call, we cannot take advantage of knowing the state of a wire as it is passed into or out of a function. Therefore, function parameters' states and return states are marked as *UNKNOWN*. It is possible to pass or return wires with "0" and "1" states, but it is not as efficient as the optimizer cannot use the information that they are "0" and "1" since they must be marked as *UNKNOWN*. This inefficiency is necessary since we only output each function a single time preventing us from taking advantage of specific parameter states. We could solve this by outputting multiple function files with different wire parameters, but this would expand the size of our circuit representation.

6.3.2 Circuit Interpreter

Using our circuit output format, the process of interpreting a circuit is reduced to a highly efficient task. When the interpreter is initially called, it reads an *.mfrig* file, which contains information about the number of parties, input and output sizes, which wires correspond to the input and output, and the number of functions. After these parameters are initialized, the interpreter is ready for the first *getNextGate* command. Each time *getNextGate* is called, the compiler reads the next instruction, opens the correct *.ffrig* function file, and issues the appropriate gate to the execution environment.

Each function occupies a specific set of wire values such that no function's wires will overlap. This enables us to have a "stack" of function calls without the need for the push and pop operations that would be required if our functions used overlapped wire addresses. This does not affect the output circuit size.

The interpreter holds a call stack of the active functions in its internal state. Each function, rather than being held completely in memory, is stored as a pointer to the active instruction. When a function is called the stack of

functions is updated, the current active function is set to the called function, and the called function is set back to the first instruction.

6.4 Procedures

While our technique of dividing programs into distinct functions and then composing the circuit with calls to those functions allows for a significant reduction in the representation size of many circuits, not all programs can be easily partitioned into distinct functions. If a clean partitioning does exist, often times the function overhead for copying parameters and return values may exceed the number of commands inside the function. This obstacle is commonly encountered with loops, where each loop iteration is copied to a separate function file, creating redundant data that expands the size of the circuit representation. To reduce the output file size in this case, we develop a novel construct which we call *procedures*. A procedure is a repeated area of a loop where the output sub-circuit is placed inside of a function. Instead of outputting the sub-circuit each time the loop iterates (i.e., unrolling the loop), all that is required for a procedure is adding a function call. Each iteration of a loop using procedures adds a negligible amount of overhead to the output file size (a single function call). Procedure calls do not require any overhead for arguments as it uses the same set of wires already defined in the calling function.

To demonstrate the output file size reduction possible using procedures, we consider an example program that adds five 32-bit variables to an accumulator 1000 times (the full program is in Appendix B). If no procedure is used, this program requires an output file of about 13MB since each iteration of the main loop must be unrolled. However, if a procedure is used, then output is one 30 KB file (main) and one 13KB function file (the procedure), a reduction of the total disk usage by over 300x. Compilation time is also reduced, as the entire procedure circuit will no longer need to be output for every iteration of a loop.

Unfortunately, procedures can only be used if each iteration through the loop uses the same wire states and addresses (i.e., we only output the sub-circuit assuming a particular set of wire states and addresses). Variables used in a procedure thus have to be located at the same wire addresses and have the same states in every iteration. This excludes loops with variables located in and modified inside the loop, as the state of the variable changes with every iteration. The simplest way to ensure correctness is to force wires to one of three states at the conclusion of an assignment statement, either *UNKNOWN*, *ONE*, or *ZERO*, then sort free wires in each iteration of a loop so they will always be allocated in the same order. Our implementation follows this technique, using the Radix sorting algorithm to keep wires in order.

7 Experiments

7.1 Frigate Correctness

To demonstrate the correctness of our compiler, we tested Frigate using the tests that we generated to examine the existing compilers in the community. After hundreds iterations of development and testing and months of work, Frigate successfully passed all correctness tests, and produces correct and functioning circuits in every case where previous compilers failed. For further detail on the state space examined in Frigate, see Appendix E.

7.2 Compiler Efficiency Tests

By constructing a compiler using our four development principles, we wanted to evaluate whether adhering to the principles we laid out would have an adverse effect on performance. We tested the time that is required to compile circuits in Frigate against the three most recent compilers we examined. Neither Fairplay or PAL give competitive compilation results, so we omit them from our benchmarks. All of our benchmarking tests were performed on a MacBook Pro with an Intel i7 4-core 2.3Ghz with 16GB RAM, 256KB L2/core, and 6MB L3.

7.2.1 Test Programs

To evaluate performance accross a wide variety of compilers, we used common test programs used by the other researchers in this space [25, 24, 40]. We used the following test programs: multiplication with matrices of X by X with 32-bit values, AES, Hamming distance of two X bit numbers, multiplication of two X-bit numbers, and RSA (modular exponentiation) of X bits, where the base, exponent, and modulus are all X bits in length. For each test program, we varied the input size X.

7.2.2 Tests

We summarize the results in Figure 4 by comparing the largest input values for each program that successfully compiled across all compilers. In every case, Frigate completes compilation the fastest. In the best case, the application Mult 256 that computes the multiplication of two 256-bit numbers, Frigate completes three orders of magnitude faster than the next fastest compiler, PCF.

In addition to comparing speed efficiency, we also considered the non-XOR gate counts of each program compiled. Because the free-XOR optimization for garbled circuits [23] allows XOR gates to be evaluated

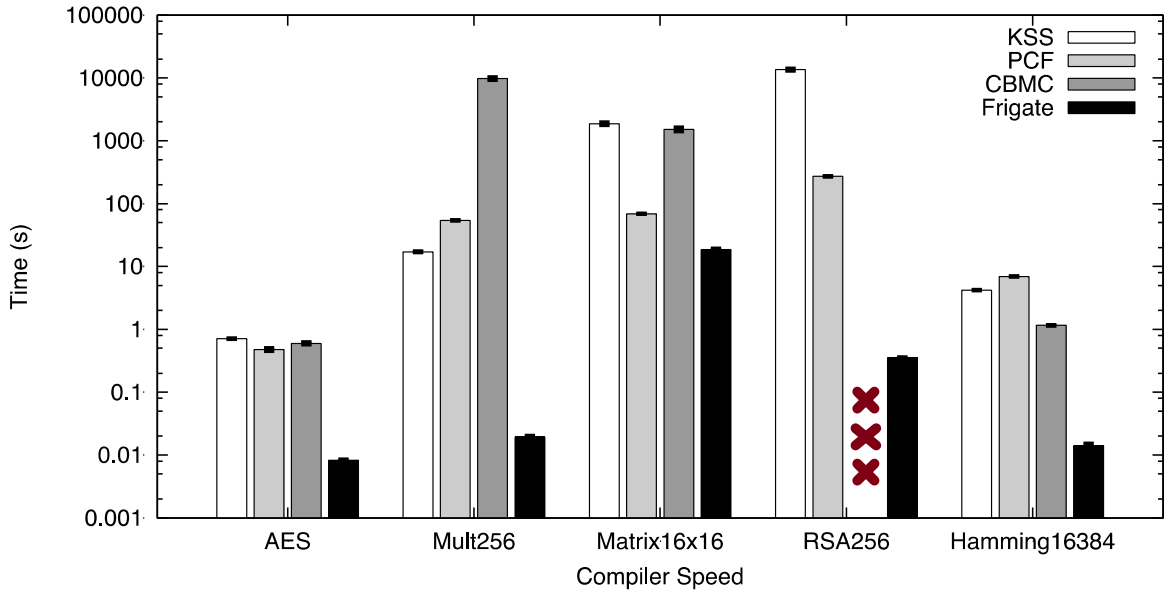


Figure 4: Comparing the different compilers we tested for compilation time. We did not succeed in compiling RSA256 with CBMC. Note the y-axis is logscale.

with non-cryptographic operations and without consuming network bandwidth, we consider non-XOR gates the bottleneck in computation. Frigate greatly reduces the number of non-XOR gates in two of the applications tested, Hamming Distance and Mult, demonstrating up to 6x reduction in the number of non-XOR gates. In the case of AES and RSA, the improvement was only slightly better than existing compilers, reducing the gate count by up to 1.11x. In only one case did we observe a reduction in gate count efficiency, Matrix Multiplication. This is due to factors such as additional overhead incurred due to Frigate's ability to handle multiplication inputs as either signed or unsigned in the same circuit, unlike other compilers. Table 4, in the Appendix C.1, gives the full results for all programs we tested.

ProgramName	PCF	Frigate	Imp.
Hamming 1000	$0.40 \pm 3\%$	$0.0040 \pm 4\%$	100x
Hamming 16384	$4.8 \pm 2\%$	$0.015 \pm 1\%$	320x
Mult 256	$2.11 \pm 0.5\%$	$0.005 \pm 20\%$	422x
Mult 4096	$280 \pm 2\%$	$0.50 \pm 8\%$	560x
Matrix Mult 5	$0.650 \pm 0.5\%$	$0.0048 \pm 4\%$	135x
Matrix Mult 16	$17.5 \pm 0.8\%$	$0.071 \pm 1\%$	246x
AES	$0.69 \pm 1\%$	$0.0008 \pm 30\%$	862x
RSA 256	$690 \pm 0.9\%$	$4.27 \pm 0.9\%$	161x
RSA 512	$4880 \pm 0.8\%$	$34.0 \pm 0.6\%$	143x

Table 2: Results from testing the PCF and Frigate interpreters and their speed. In this set of tests we only simulate the execution of the circuit. The *imp.* column shows how many times faster Frigate's interpreter is compared with the PCF interpreter. All times in seconds.

7.3 Interpreter and Execution Speed

Interpreter Time: Our next set of experiments compares the performance of the Frigate and PCF interpreters. Table 2 shows our experimental results. The Frigate output format allows for significant reduction in interpreting time. In the worst-case, we improve over PCF by 45x, with a reduction of 682x in the best case. To demonstrate the practical ramifications of interpreter time, we give a brief analysis to show how Frigate is able to improve the overall performance of a garbled circuit protocol.

Simulated Protocol Time: The cost to generate a garbled-circuit in a protocol can be expressed as:

$$InterpreterTime + non\_XORgates * timeToGarbleAGate$$

In the above equation, $InterpreterTime$ is the time to read each gate from the circuit file, $non\text{-}XORgates$ is the number of non-XOR gates in the circuit, and $timeToGarble$ is the cost for garbling and sending a single non-XOR gate. $InterpreterTime$ also includes a single payment for the cost of the free operations. To gather the values used in our analysis, we experimented with an efficient semi-honest garbled circuit implementation based on Kreuter et al. [25] that could generate and send approximately 1.2 million non-XOR gates in a second in the best case (833 ns per gate). This time could be further reduced by applying recent optimizations such as the fixed key block cipher technique developed by justGarble [7].

Consider the example program Mult 256. In Frigate, the total interpreter time is about 0.005 seconds with 65543 non-XOR gates. With PCF, the total interpreter time is about 2.11 seconds with 400,210 non-XOR gates. Using these values, we can define the total time to garble the circuit in Frigate as $0.005 + 65,543 * .000000833$ for a total of 0.0596 s. For PCF, the total time is approximately $2.11 + 400,210 * .000000833 = 2.44$ s. This means Frigate's expected runtime is better by approximately 40x. Considering this simple program, the bottleneck for garbling using both Frigate and PCF is the interpreter time and not the time required to garble gates. When we extend this analysis to protocols secure against malicious adversaries, another challenge arises. This security model requires a cut-and-choose operation where the circuit is garbled multiple times. In PCF, the interpreter is designed to read the circuit file once for each circuit in the cut-and-choose (including free operations). A more efficient design that we adopt is to read the circuit file once and garble each gate multiple times from memory. Using 80 circuits for $\frac{1}{2^{80}}$ security (using Lindell's fast cut-and-choose [45]) means we have a total runtime of $0.005 + 0.01 * 0.43 * 79 + 65543 * .000000833 * 80$ or 4.7s for Frigate, since the garbling time and not the interpreter time, is multiplied by the number of circuits produced. We include an added term, .43, of the total interpreter time for each circuit in Frigate as we found executing operations took about 43% of the total time. With PCF, the expected time is $2.5 * 80 + 400,210 * .000000833 * 80 = 226.7$ s. This gives us an expected improvement of about 48x over PCF. Given this observation, having an efficient interpreter can significantly affect the speed of a garbled circuit protocol.

Appendix C.2 lists the expected efficiency for all of our test programs using the formulas introduced above.

7.4 Discussion

Extensibility: After the initial creation and implementation of Frigate we made an additional change in order to show extensibility. We enabled constants to be defined to a specific bit-length, which was not in our original specification allowing negative constants to be correctly assigned. For developers to extend Frigate with their own functionality, they simply create or modify an AST node and the parsing rules, modify typing for new or existing operators, and then define what sub-circuit the operator outputs.

Tools: To demonstrate how Frigate can be used to create useful developer tools, we created an extension to output the gate counts of program components inline in a printout. We implemented this tool specifically as it is important to understand where the most costly gate operations are in a Boolean circuit program. Our tool also maps in the cost of function calls (and procedures) even though they are not called during compilation. Figure 5 shows an example program and its gate counts. The numbers in the angle brackets are $\langle non{-}XOR\ gates, free\ operations \rangle$.

```
:
#define wiresize 1024 #parties 2
typedef int 1024 int
#input 1 int #output 1 int
#input 2 int #output 2 int
function int mul(int x, int y)
<1048581,2094078>{
        return x * y;
}
function void main()
<270534921,541326327>{
        int t = input1;
        for(int i = 0; i < 256; i++)
        <268436736,537132800>{
                t = mul(t, input1);
        }
        <1048581,2094078>{
                t = t * input1;
        }
        output1 = input1 * input2 + t;
}
compiler: time(s): 0.877391
interpreter: gates: nonxor: 270534923
free ops: 541330424 time(s): 3.63717
```

Figure 5: Example of Frigate's gate counts in the program at each compound statement.

8 Related Work

When the garbled circuit protocol was developed by Andrew Yao [44], it was one of the first protocols to demonstrate that secure multiparty computation was possible. However, the protocol remained a theoretical novelty until the Fairplay implementation [30] demonstrated that the protocol could be feasibly run for small sized circuits. In more recent work, the garbled circuit protocol has been vastly expanded from its original capability, with protocols allowing for multiple parties [8], security in the presence of covert [14], malicious [21, 27, 28, 39, 40], and other adversaries [19], as well as outsourced execution from computationally limited devices [20, 10, 9]. However, one of the major remaining limitations is the size of the garbled circuit representation, which prevents very large functions from being executed practically due to the amount of bandwidth required to transmit the circuit, as well as the computation time to evaluate it.

To help reduce the size of the garbled circuit, several protocol optimizations have been developed. The free-XOR technique [23, 12] allows for garbled XOR gates to be evaluated with a single XOR operation, and require zero bandwidth to transmit. In addition, optimizations

such as garbled row-reduction [35] allow for the size of the transmitted AND gates to be reduced by a constant factor. Other optimizations, such as FleXOR [22], have been shown to reduce bandwidth and computation time for certain functions. The pipelining technique developed by Huang et al. [18] generates and transmits the circuit in layers, allowing large circuits to be handled in a small amount of memory. Most recently, the PartialGC system [31] allows for garbled wire values to be re-used between protocol executions, reducing the number of consistency checks required to ensure security. However, while these protocol optimizations allow for constant factor improvements in speed and bandwidth, they do not optimize the size of the boolean representation itself, and are limited in their ability to make very complex functions execute in a practical amount of time.

The first system to compile a high-level function into a boolean circuit representation for secure computation was the Fairplay compiler [30]. While this compiler provided a first step towards a practical and usable means for representing arbitrary programs as circuits, the Secure Function Definition Language (SFDL) was very limited in its complexity and ability to represent programs. Furthermore, the compiler produced large, unoptimized circuits, and suffered from a number of correctness issues. To reduce the size of the unoptimized circuit representation, the PAL compiler [32] introduced the concept of templates, which allowed simple functionalities within a circuit to be represented by a template rather than a repeated set of gates. The first compiler to incorporate a number of circuit optimizations to reduce the actual gate count was by Kreuter, shelat, and Shen [25]. The Portable Circuit Format compiler (PCF) [24] combined the concept of templating with several circuit optimizations that were both novel or derived from the KsS compiler. However, the interpreting environment of PCF causes an increase in execution time, and stability issues have significantly reduced the practical usability of the compiler.

9 Conclusion

Garbled circuit protocols have made significant advances based upon the development of a set of circuit compilers that allow researchers to quickly develop new test applications. However, the error-prone nature of these compilers has made building new research on them questionable. In this work, we examine the state of secure computation compilers using rigorous validation testing. From this examination, we present a set of guiding principles for secure computation compiler design, and develop the Frigate compiler based on these principles. By building a principled compiler and thoroughly validating correctness, our compiler reduces compile time by as much as three orders of magnitude when compared to previous compilers. Furthermore, our novel circuit representation format allows for circuit interpretation time to be reduced by as much as 600x. These results demonstrate that a principled approach to design and validation of secure computation compilers produces tools that are both correct and efficient, and offer the rest of the community a solid foundation on which to develop further research.

References

[1] Arm Compiler Verification Process. http://www.arm.com/products/tools/software-tools/mdk-arm/compilation-tools/compiler-verification.php.

[2] flex: The Fast Lexical Analyzer. http://flex.sourceforge.net.

[3] Gnu bison. http://www.gnu.org/software/bison/.

[4] SuperTest Compiler Test and Validation Suite. http://www.ace.nl/compiler/supertest.html.

[5] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2006.

[6] I. D. ard, V. Pastro, N. P. Smart, and S. Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology - Crypto*, 2012.

[7] Bellare, Mihir and Hoang, Viet Tung and Keelveedhi, Sriram and Rogaway, Phillip. Efficient Garbling from a Fixed-Key Blockcipher. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (Oakland '13)*, 2013.

[8] A. Ben-David, N. Nisan, and B. Pinkas. FairplayMP: a system for secure multi-party computation. In *Proceedings of the ACM conference on Computer and Communications Security*, 2008.

[9] H. Carter, C. Lever, and P. Traynor. Whitewash: Outsourcing garbled circuit generation for mobile devices. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2014.

[10] H. Carter, B. Mood, P. Traynor, and K. Butler. Secure outsourced garbled circuit evaluation for mobile devices. In *Proceedings of the USENIX Security Symposium*, 2013.

[11] Cheng Wang and Rengan Xu and Chandrasekaran, S. and Chapman, B. and Hernandez, O. A Validation Testsuite for OpenACC 1.0. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, 2014.

[12] S. G. Choi, J. Katz, R. Kumaresan, and H.-S. Zhou. On the security of the "Free-XOR" technique. In *Proceedings of the International Conference on Theory of Cryptography*, 2012.

[13] Garoche, Pierre-Loïc and Howar, Falk and Kahsai, Temesghen and Thirioux, Xavier. Testing-Based Compiler Validation for Synchronous Languages. In *NASA Formal Methods*. 2014.

[14] V. Goyal, P. Mohassel, and A. Smith. Efficient two party and multi party computation against covert adversaries. In *Proceedings the annual international conference on Advances in cryptology*, 2008.

[15] T. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, Jan. 2003.

[16] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith. Secure Two-party Computations in ANSI C. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, 2012.

[17] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS '12: Proceedings of the 19th ISOC Symposium on Network and Distributed Systems Security*, San Diego, CA, USA, Feb. 2012.

[18] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *Proceedings of the USENIX Security Symposium*, 2011.

[19] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[20] S. Kamara, P. Mohassel, and B. Riva. Salus: A system for server-aided secure function evaluation. In *Proceedings of the ACM conference on Computer and communications security (CCS)*, 2012.

[21] M. S. Kiraz. *Secure and Fair Two-Party Computation*. PhD thesis, Technische Universiteit Eindhoven, 2008.

[22] V. Kolesnikov, P. Mohassel, and M. Rosulek. FleXOR: Flexible Garbling for XOR Gates That Beats Free-XOR. In *Advances in Cryptology – CRYPTO*, 2014.

[23] V. Kolesnikov and T. Schneider. Improved Garbled Circuit: Free XOR Gates and Applications. In *Proceedings of the international colloquium on Automata, Languages and Programming, Part II*, 2008.

[24] B. Kreuter, B. Mood, a. shelat, and K. Butler. PCF: A Portable Circuit Format for Scalable Two-Party Secure Computation. In *Proceedings of the USENIX Security Symposium*, 2013.

[25] B. Kreuter, a. shelat, and C.-H. Shen. Billion-gate secure computation with malicious adversaries. In *Proceedings of the USENIX Security Symposium*, 2012.

[26] X. Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7), July 2009.

[27] Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *Proceedings of the annual international conference on Advances in Cryptology*, 2007.

[28] Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Proceedings of the conference on Theory of cryptography*, 2011.

[29] Lopes, NunoP. and Monteiro, José. Weakest Precondition Synthesis for Compiler Optimizations. In *Verification, Model Checking, and Abstract Interpretation*, 2014.

[30] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay–a secure two-party computation system. In *Proceedings of the USENIX Security Symposium*, 2004.

[31] B. Mood, D. Gupta, K. Butler, and J. Feigenbaum. Reuse it or lose it: More efficient secure computation through reuse of encrypted values. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, 2014.

[32] B. Mood, L. Letaw, and K. Butler. Memory-efficient garbled circuit generation for mobile devices. In *Proceedings of the IFCA International Conference on Financial Cryptography and Data Security (FC)*, 2012.

[33] Namjoshi, KedarS. and Tagliabue, Giacomo and Zuck, LenoreD. A Witnessing Compiler: A Proof of Concept. In *Runtime Verification*. 2013.

[34] G. C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, May 2000.

[35] B. Pinkas, T. Schneider, N. P. Smart, and S. C. Williams. Secure Two-Party Computation is Practical. In *ASIACRYPT*, 2009.

[36] Plum Hall, Inc. The Plum Hall Validation Suite for C. `http://www.plumhall.com/stec.html`.

[37] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, London, UK, 1998.

[38] T. Schneider and M. Zohner. GMW vs. Yao? Efficient Secure Two-Party Computation with Low Depth Circuits. In *Financial Cryptography and Data Security*, 2013.

[39] a. shelat and C.-H. Shen. Two-output secure computation with malicious adversaries. In *Proceedings of EUROCRYPT*, 2011.

[40] a. shelat and C.-H. Shen. Fast two-party secure computation with minimal assumptions. In *Conference on Computer and Communications Security (CCS)*, 2013.

[41] J.-B. Tristan, P. Govereau, and G. Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, 2011.

[42] Wang, Cheng and Chandrasekaran, Sunita and Chapman, Barbara. An OpenMP 3.1 Validation Testsuite. In *OpenMP in a Heterogeneous World*, 2012.

[43] Yang, Xuejun and Chen, Yang and Eide, Eric and Regehr, John. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*, 2011.

[44] A. C. Yao. Protocols for secure computations. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1982.

[45] Yehuda Lindell. Fast Cut-and-Choose Based Protocols for Malicious and Covert Adversaries. In *Advances in Cryptology (CRYPTO '13)*, 2013.

A Operator Typing

In Figure 6 we give the full typing rules for our language. Each operator or statement has its own typing rule.

Each of the typing rules has a set of types on the top i.e., $t : T$ means t is of type T in the program context Γ. The bottom is a statement in Frigate's input, along with the types of each item. *Num* is a number of any bit-length. Num_{L_i} is a number of bit-length i. R, T, F, S, P can be replaced with any type but use different names to differentiate where they are used.

B Example Programs

Input Example

The program below gives an example of much of the syntax in Frigate's input language, using different statements, declaring types, and setting input and output types. This main function simply sets the both output values to be the addition of the two input values.

```
#define wiresize 32
#parties 2

typedef int_t wiresize int

typedef struct_t mystruct {
    int x;
}

typedef struct_t newstruct {
    int x;
    newstruct var[5];
}

#input 1 int
#output 1 int
#input 2 int
#output 2 int

function void main()
{
    output1 = input1 + input2;
    output2 = input1 + input2;
}
```

Procedure Example

This is the example program discussed in Section 6.4

```
function void main()
{
    int x = input1;
    int y = input2;

    for(int i=0;i<1000;i++)
    {
        x = x + y + y + y + y + y;
    }
    output1 = x;
}
```

	Security	
Program	Semi-Honest	Malicious
Hamming 1000	73x	124x
Hamming 16384	147x	198x
Mult 256	40x	43x
Mult 4096	24x	24x
Matrix Mult 5	6x	6x
Matrix Mult 16	5x	5x
AES	74x	78x
RSA 256	4x	4x
RSA 512	4x	4x

Table 3: This table shows the expected speedup Frigate has over PCF for all programs in both the semi-honest and malicious setting using the formulas in the experiments section.

C Extended Performance Results

C.1 Expanded Compilation Time Result Table

Table 4 shows the complete compile times for each program and the compiler combination.

C.2 Expected Performance

In Table 3, we give the expected performance results for each program using the formulas from the experiments section.

D Frigate Design Details

This section lists some of our design decisions and explains why the decided to do things differently from other compilers.

D.1 No Recursion

We do not allow recursion in our execution model. Multiple copies of a specific function could be created to simulate recursion but this is not done as part of a native operation. Since the depth of recursion must be known at compile time, this does not remove functionality from the language but may reduce expressiveness.

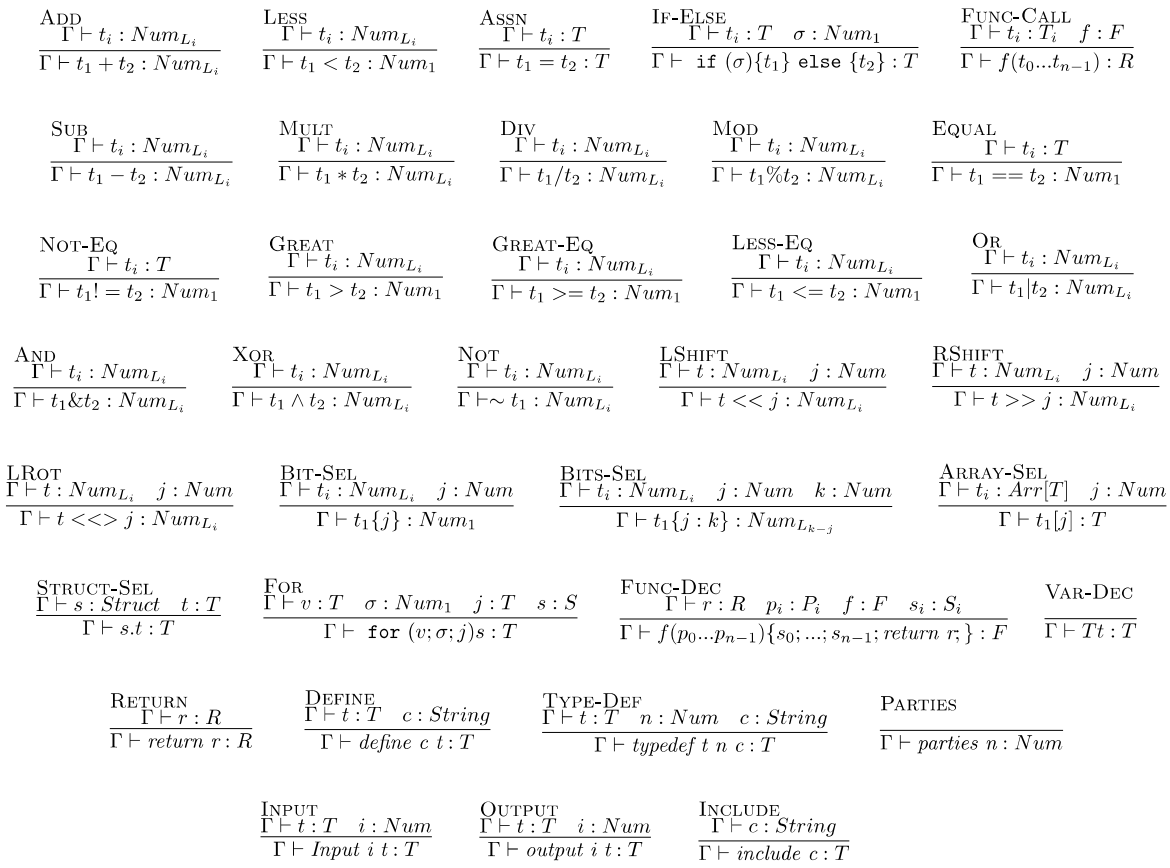


Figure 6: The full set of typing rules for all Frigate operators.

D.2 void Functions

Our compiler allows functions that return nothing. At first this may seem counter intuitive to our model since nothing performed in these functions would ever be used. However, extensions like adding the ability to pass variables to functions by "reference" may require the use of *void* functions. We therefore included them into our compiler.

D.3 Unsigned Division

Unlike our other operations where the user determines how to interpret the results after the circuit is evaluated, we only have unsigned division and unsigned remainder division, since the user must indicate if it was a signed or unsigned operation. Declaring the sign of types is a possible extension for future work.

Process to use unsigned division operator for signed division The process for performing signed division with an unsigned division operator is as follows: check the signs of both operands, if either is less than 0 then negate the operand so it is positive and save that the operand was negative. Execute the unsigned division with the positive values. If one of the original values was negative then negate the result, otherwise leave the value as positive.

D.4 No Sign Extension By Default

We do not sign extend our operations. However, we (ex post facto) added a technique to allow constants to have a set bit-length to prevent the need for sign extension, i.e. -1, instead of being a 2-bit number (constants are sized to their bit-length + 1 by default) could be defined as 8 bits. To this end we added the # operator to specify the bit-length of constants (i.e. "*char16 x = -9#16;*" defines -(9) to be 16 bits in length instead of 5 bits in length. If we did not specify the bit length and assign -9 is to x, -9 would be lost).

D.5 More Than Two Parties

Our compiler allows more than two parties in the computation unlike the other compilers we examined. Adding

	Time(s)	All Gates	Non-XOR	Time(s)	All Gates	Non-XOR
Program	Frigate			PCF		
Hamming 1000	0.0092 ± 3%	7,402	1,803	5.1 ± 1%	21,970	4,882
Hamming 16384	0.014 ± 8%	89,245	21,642	6.95 ± 0.8%	391,683	96,117
Mult 256	0.019 ± 5%	195,334	65,543	54.2 ± 0.7%	1,659,808	400,210
Mult 4096	3.9 ± 2%	50,311,174	16,777,223	63.7 ± 0.9%	364,605,460	89,444,609
Matrix Mult 5	0.092 ± 4%	383,877	132,502	60.2 ± 0.4%	433,475	127,225
Matrix Mult 16	19 ± 4%	12,578,818	4,341,762	68.8 ± 0.4%	14,308,864	4,186,368
AES	0.0082 ± 3%	33,794	10,258	0.48 ± 5%	38,260	12,578
RSA 256	0.355 ± 0.1%	1,008,796,419	219,154,946	272 ± 0.6%	673,105,990	235,925,023
RSA 512	1.38 ± 0.5%	8,061,715,971	1,749,029,890	275 ± 0.8%	5,397,821,470	1,916,813,808
Program	CBMC			KSS		
Hamming 1000	0.71 ± 2%	54,233	18,906	2.2 ± 1%	20,493	4,641
Hamming 16384	1.16 ± 0.8%	910,495	290,728	4.21 ± 0.8%	370,110	88,952
Mult 32	0.48 ± 1%	6,223	1,741	0.34 ± 6%	15,935	5,983
Mult 256	9,800* ± 5%	5,880,833	2,264,860	17 ± 2%	1,044,991	391,935
Matrix Mult 5	1.8 ± 2%	795,988	223,720	32 ± 2%	1,968,452	746,177
Matrix Mult 16	1,500* ± 7%	26,182,494	7,251,991	1900 ± 5%	64,570,969	24,502,530
AES	0.60 ± 4%	35,607	11,469	0.71 ± 1%	49,912	15,300
RSA 256**	-	-	-	14,000 ± 4%*	928,671,864	315,557,288

Table 4: This table shows the compile time in seconds, the total gates, the non-XOR gates and the total operations (specific to PCF and Frigate where there are additional operations besides gates). Note that for CBMC and KSS, we ran Mult 32 and Mult 256 instead of Mult 256 and Mult 4096.
All tests were ran 10 times unless otherwise noted: * tests ran 3 times, ** we stopped trying to compile this program after 6 hours.

additional parties to the computations can be useful to declare different types of output.

E Frigate Validation Details

This section details our validation tests for a correct secure computation compiler. To properly validate a compiler we have to check all possible ways that each statement can output a sub-circuit and check the ways data can flow from the beginning to the end of the program (including when the data is encapsulated in variables, when it is used in control structures, etc.). While daunting, the task is made simplilar once the realization is made that each operator and control structure can only be output in a finite number of ways, i.e., an *if/else* statement has 2 possibilities: it is either the first *if/else* statement or is nested under at least one other *if/else* statement.

We perform the tests outlined in Section 3. At the conclusion of these tests we have covered the state space in Frigate. We have tested (1) the correctness of each mini-circuit an operator uses, (2) all the ways in which data can populate each operator, (3) the base and nested rule for each construct (*if* statements, *for* loops, and arrays declarations), (4) common edge cases, and (5) unique constructs to Frigate's input format.

For some tests in Frigate, like verifying whether a file is included correctly, were performed by printing out the AST and not by compiling and then executing the programs.

For each type of test, we test a variety of positive (correct) and negative (incorrect) results with emphasis on edge cases. As an example, here are the test cases for the addition operator:

1. Do a variety of different input value combinations give correct results?
2. Can it handle negative numbers?
3. Does adding two 'unsigned' positive numbers that overflow into the sign bit give the correct result or does having signed addition produce errors?
4. Does adding two different types of the same length give a warning?
5. Does adding two different types of different lengths give an error?

Operators: The first tests on the operators examine whether the sub-circuits, or templates, for each operator (adder, subtractor, etc.) are correct. It is easy to see whether these tests are correct by outputting the result or by an examination of the output for simple sub-circuits.

Once we know the template circuits are correct, we must then show all possible types of data that can be entered into the template work as well. These types are: (1) constants, (2) variables, or (3) results from an expression.

```
if(x) {
if(y) { } else { }
}
else {
if(z) { } else { }
}
```

Figure 7: Twice nested *if* statements. There are 8 possible combinations as x, y, and z can either be 0 or 1.

Once these tests pass, we know the operator correctly takes in all the different types of data.

Control Structures: Once each operator is shown to be correct, we know any other errors found will not be from the primitive operators but from the control structures. There are four control structures in Frigate we must test: functions, *if/else* statements, *for* loops, and procedures.

For each control structure, we check every way in which it can be output. Conditional *if/else* statements can be checked for correctness by performing an exhaustive search up to depth 2 (i.e., test all 8 possible cases of the conditional output as shown in Figure 7). By selecting depth 2, the unique ways to output the *if/else* statement occur within the first conditional, while the depth-2 *if/else* statment must be combined with its parent conditional. If the nested *if/else* conditionals combine correctly at depth 2 then by induction, it will work for subsequently nested conditionals as well. Each *if/else* statement should be tested for when the guard values are dependent on user input as well as when they are not dependent on user input.

It is relatively simple to validate the correctness of *for* loops when they are only nested under another *for* loop by checking whether they output the circuit the correct number of times. When they are used under *if/else* statements, problems can arise depending on how the loop variable is scoped and whether the loop variable's result will be labeled *UNKNOWN*, meaning the result is based on user input due to the *if* statement, or whether it will be labeled as a 0 or 1 value. We test to depth 2 in case there is an external state used by the compiler that may prevent nested *for* loops from working correctly.

Functions also have a finite number of possible states to test. Our procedure for carrying out this testing was as follows. (1) Test the function call operator, where a function call is treated as any other operator that takes in any number of operands (parameters) and returns a single operand. We test different possible combinations of parameters up to length 2, as that is where data no longer acts in a unique way. (2) Function definitions need to be tested to ensure different types of return variables (array, struct, int) work correctly. (3) Test two of the same function call in an operand with different results (e.g., `addX(3,4) + addX(5,6)`). It is possible the results of the first call may be overwritten by the second call. (4) Test that parameters can be used inside of the functions.

Although we do not have global variables in our input language, we suggest the following two tests for compilers that use global variables. (1) Test whether functions correctly modify global variables. (2) Test when functions are called under an *if/else* statement and whether the function modifies the global variable as expected, i.e., if the guard is *false* then the global variable is not modified.

The correctness of procedures reduces to a simple question: *is each variable composed of the exact same wires every iteration?* We know that variables will use the same free wires if the wire pool is sorted before each iteration. Procedures can be difficult to use and if procedures are not used correctly by a developer, then a program will output undesired results. However this is not a *correctness* problem.

We test various specific cases: Empty functions, array declarations and access up to 2 dimensions (depth 2 as each dimension is compartmentalized, if it works for depth 1 and depth 2, then it works for the base case and when 'nested'), verify that *#includes* and *#defines* work correctly, check that the *parties* variable correctly determines what input and output variables can be used, assignments of arrays and structs, as well as the additional edge cases mentioned above.

Frigate's Interpreter: In order to use the circuits generated by Frigate, we also validate the correctness of the interpreter. If the interpreter was created first in order to test the functionality of the compiler then it will correctly function after all the functionality of the compiler was checked (i.e., there are no more edge cases to check where it would fail). If the interpreter was not created first then each test should be run through the interpreter.

Glossary of Terminology

AES: Advanced Encryption Standard

Blackbox: A means of viewing a cryptographic protocol such that it can be viewed as secure as a whole without necessarily proving the composition of its components (which are known to be secure).

CBMC: A Garbled Circuit Compiler

CMTB Outsourcing: Built on the KSS framework, the CMTB evaluation framework outsources the evaluation of garbled circuits to allow mobile devices to participate more efficiently in garbled circuit computation.

EMOC: Efficient Mobile Oblivious Computation: EMOC is a set of protocols using partially homomorphic cryptosystems to compare encrypted elements.

Evaluator: The party responsible for running a garbled circuit.

Fairplay: A Garbled Circuit Compiler

Garbled circuit: A scrambled representation of a low-level circuit designed to prevent its evaluator from understanding the inputs or outputs.

Generator: The party responsible for garbling a circuit and inputs.

GPS: Global Positioning System

IR: Intermediate Representation

KSS: A Garbled Circuit Compiler, The KSS compiler and evaluation framework combines a number of garbled circuit optimizations into a malicious secure protocol for two-party garbled circuit evaluation.

MAC: Message Authentication Code

OBDD: Ordered Binary Decision Diagrams

OT: Oblivious Transfer: A cryptographic protocol wherein a user is able to learn 1 out of n secret values held by a server, and the server is unable to determine which value the user learned.

PAL: Pseudo Assembly Language: A Garbled Circuit Compiler, The PAL compiler uses circuit templates to save memory when compiling garbled circuits on a resource-constrained platform, such as a mobile device.

PCF: Portable Circuit Format, A Garbled Circuit Compiler

RSA: Rivest-Shamir-Adleman Cryptosystem: A Public-Key Cryptosystem based on two large prime numbers and an auxiliary value.

SFE: Secure Function Evaluation: A protocol by which a function can be evaluated such that its inputs remain private, and only its output is exposed.

Whitewash Outsourcing: Whitewash builds on the Shelat-Shen protocol (CCS 2013) for garbled circuit SMC, allowing mobile devices to securely and efficiently outsource the costly operations associated with circuit garbling.